НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Кафедра автоматизованих систем обробки інформації і управління

«На правах рукопису»

УДК _____

«До захисту допущено»
В.о. завідувача кафедри
_____ *О.А.Павлов*
(підпис) (ініціали, прізвище)
“___”_____2019 р.

Магістерська дисертація

зі спеціальності *121 «Інженерія програмного забезпечення»*

на тему: «*Архітектура програмного забезпечення для пошуку подій* »

Виконав : студент VI курсу, групи *ІП-82мп*

Стеценко Владислав Олегович _____
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник *доц., к.т.н., Ліщук К.І.* _____
(посада, науковий ступінь, , вчене звання, прізвище та ініціали) (підпис)

Консультант *доц., к.т.н., Ліщук К.І.* _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент *доц., к.т.н., Ткач М.М.* _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.

Студент _____(підпис)

Київ – 2019 року

РЕФЕРАТ

Актуальність теми: Актуальність дослідження зумовлюється неабияким ростом зацікавленості бізнесу в швидких, легко масштабованих сервісах, що в свою чергу дозволило б зменшити витрати на розробку веб-застосунків та прискорити її. Дослідження особливостей мікросервісної архітектури допоможе зрозуміти та виділити всі її плюси та мінуси, а також дослідити шляхи вирішення ключових проблем. Ці дослідження дозволять розробити прототип вебсистеми, що буде вирішувати купу проблем мікросервісів та надавати розробникам можливість швидко розпочинати роботу над мікросервісним проектом. Оскільки для дослідження потрібно розробити веб-сервіс, було обрано тематику веб-афіш.

Мета дослідження: основна мета роботи полягає в дослідженні та розробці архітектури программного забезпечення для пошуку подій.

Для реалізації поставленої мети були сформульовані **наступні завдання:**

- аналіз відомих на даний момент архітектурних рішень для побудови программного забезпечення для пошуку подій;

- аналіз відомих на даний момент методів та підходів по побудові мікросервісної архітектури;

- розробка методики побудови программного забезпечення на основі мікросервісів, який би дозволяв створювати програмне забезпечення, котре буде відповідати усім вимогам замовника;

- вибір необхідних програмних засобів для розробки программного забезпечення;

- розробка программного забезпечення на основі мікросервісів, використовуючи запропоновану архітектуру;

- дослідження ефективності запропонованого архітектурного рішення.

Об’єкт дослідження: процес розробки програмного забезпечення для пошуку подій.

Предмет дослідження: архітектурні рішення, котрі можуть використовуватися для розробки программного забезпечення для пошуку подій.

Наукова новизна:

Найбільш суттєвими науковими результатами магістерської дисертації є:

- запропоновано архітектурне рішення для побудови программного забезпечення для пошуку подій на основі мікросервісної архітектури.

Практичне значення отриманих результатів визначається тим, що методику запропонованого архітектурного рішення побудови программного забезпечення доведено до практичної реалізації. Запропонована архітектура програмного забезпечення надає можливість розробникам уникнути ключових недоліків робот из мікросервісами і разом з тим пришвидчити процесс розробки самого программного забезпечення.

Зв'язок з науковими програмами, планами, темами: робота виконувалась на кафедрі автоматизованих систем обробки інформації і управління Національного технічного університету України "Київський політехнічний інститут імені Ігоря Сікорського".

Публікації: Стеценко В.О. Спосіб декомпозиції систем з використанням мікросервісно архітектури. / Стеценко Владислав Олегович. // The scientific heritage. – 2019. – №41. – С. 45-47.

КЛЮЧОВІ СЛОВА: МІКРОСЕРВІСНА АРІТЕКТУРА, МОНОЛІТНИЙ ЗАСТОСУНОК, СЕРВІСНО-ОРІЄНТОВАНА АРХІТЕКТУРА.

ABSTRACT

Topicality: The relevance of the study is driven by the growing interest of businesses in fact, easily scalable services, which in turn would reduce the cost of developing web-applications and accelerate it. Investigating the features of a microservice architecture will also help you understand and highlight all its pros and cons, a well as explore ways to solve key problems. These studies will help to develop a prototype of the web system that will solve a a bunch of microservice problems and give developers the opportunity to get started on a microservice project quickly. As a web service was required for the study, the topic of the web poster was selected.

Purpose of the study: the main purpose of the work is to research and develop an event-finding software architecture.

To achieve this goal, the following tasks were formulated:

- analyze currently known architectural solutions for building a web application for event searching;

- analyze currently known methods and approaches for building a microservice architecture;

- development of a microservice-based software development methodology that would allow the creation of software that would meet all customer requirements;

- selection of necessary software tools for software development;

- development of software based on microservices using the proposed architecture;

- study of the effectiveness of the proposed solution.

Object of research the process of developing event-finding software.

Subject of research: architectural solutions that can be used to design event-finding software.

The scientific novelty: The most significant scientific results of the master's study are:

- an architectural solution for the construction of event-finding software based on microservice architecture is proposed.

The practical significance of the obtained results: is determined by the fact that the methodology of the proposed architectural solution for the construction of software is brought to practical implementation. The proposed software architecture allows the developers to avoid the key disadvantages of working with microservices and at the same time speed up the process of software development.

Relationship with scientific programs, plans and themes: The work was carried out at the Department of Computer-Aided Management And Data Processing Systems of the National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute".

Publications: Stetsenko V.O. Method of system decomposition using microservice architecture. / Stetsenko Vladyslav Olehovych. // The scientific heritage. – 2019. – #41. – P. 45-47.

КЛЮЧОВІ СЛОВА: microservice architecture, monolatic application, service-oriented architecture.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД	Електронно обчислювальна машина
СУБД	Система управління базами даних
WWW	Всесвітнє павутиння
JS	Мова програмування JavaScript
HTML	Мова розмітки гіпертекстових посилань
CSS	Каскадні таблиці стилів
IDE	Інтегроване середовище розробки
API	Прикладний програмний інтерфейс
DTO	Data Transfer Object
HTTP	Протокол передачі гіпер-текстових документів
URL	Уніфікований локатор ресурсів
TCP	Протокол кервування передачею
PEP	Policy Enforcement Point
UAA	User Authentication and Authorization
REST	Representational State Transfer
AMQP	Advanced Message Queuing Protocol

ВСТУП

Сьогодні більшість інформації, що отримує людина, знаходиться в Інтернеті. Інтернет надає велику кількість послуг:

- групи новин;
- електронна пошта та поштова розсилка;
- віддалений доступ до даних інших комп'ютерів;
- сеанси зв'язку з іншими комп'ютерами;
- пошук інформації;
- доступ до системи WWW.

В свою чергу інформаційна система WWW містить в собі величезну кількість різноманітних сервісів, таких як інтернет-магазини, соціальні мережі, електронні енциклопедії тощо. Всі ці сервіси можуть бути розроблені величезною кількістю способів та за допомогою незчисленої кількості мов програмування, бібліотек та фреймворків. На разі існує два найпопулярніших підходи до архітектури веб-застосунку: монолітні застосунки та мікросервіси. Архітектура веб-застосунку побудованого на основі мікросервісів виглядає так: є зовсім різні, незалежні одна від одної, розгорнуті зовсім окремо програми. Перша відповідає за реєстрацію, логін і збереження користувачів, друга — за створення, редагування статей, третя за коментарі і так далі. Клієнтська частина інтерфейсу користувача сама вирішує куди їй слати запити, а якщо якомусь сервісу знадобляться дані іншого сервісу, то він запитає ці дані у нього. Перша перевага полягає в тому, що для передачі даних можливо використовувати і http, і REST, і AMQP, і що завгодно . Кожен сервіс може бути написаний будь-якою мовою (тому двоє розробників будуть менше плутатись під ногами один в одного, як при розробці монолітних застосунків). І більш того кожен сервіс може мати свою базу даних. Плюс з’являється можливість легше і

швидше масштабувати додатки при збільшенні трафіку. Коли монолітні застосунки обробляють всі запити в одній програмі і піддаються лише горизонтальному масштабуванню, мікросервісний архітектурний стиль використовує підхід, при якому єдиний додаток будується як набір невеликих сервісів, кожен з яких працює у власному середовищі і взаємодіє з іншими використовуючи легковагові механізми, як правило такі як HTTP, що відкриває можливість до простішого вертикального масштабування. Ці сервіси побудовані навколо бізнес-потреб і розгортаються незалежно з використанням повністю автоматизованої середовища. Централізоване управління цими сервісами вимагають мінімум зусиль. А для розробки цих сервісів можливо використовувати абсолютно різні мови технології збереження даних. Серед цієї величезної кількості сервісів, що розміщені на просторах WWW знаходяться й так звані сайти-афіші. Здавна афішами були певні аркушеві видання, що містили в собі інформацію щодо певного культурного заходу. Головною характеристикою афіші є безпосередня передача інформації.

З розвитком Інтернету афіші переросли зі звичайних паперових видань з картинками до веб-сервісів, що мають зручний інтерфейс та велику кількість можливостей. Веб-сервіси афіші надають користувачу можливість швидко та в зручній формі знайти необхідну подію, сортувати події за типом, датами, тощо.

Актуальність дослідження зумовлюється неабияким ростом зацікавленості бізнесу в швидких, легко масштабованих, сервісах, що в свою чергу дозволило б зменшити витрати на розробку вебзастосунків та прискорити її. Дослідження особливостей мікросервісної архітектури також допоможе зрозуміти та виділити всі її плюси та мінуси, а також дослідити шляхи вирішення ключових проблем. Оскільки для дослідження потрібно розробити вебсервіс, було обрано тематику веб-афіш.

1 АНАЛІЗ ПРЕДМЕТНОГО СЕРЕДОВИЩА

1.1 Вступ

Метою даної роботи є дослідження архітектури розробки програмного забезпечення з використанням мікросервісної архітектури на прикладі вебсервісу афіші. Розроблювана система має надавати користувачам зручний інтерфейс доступу до подій, що проходять в місті. Оскільки ціллю роботи є саме огляд мікросервісів як архітектурного рішення, система має складатися з декількох незалежних вебсервісів, кожен з яких має відповідати за свої конкретні задачі та мати змогу працювати окремо від інших. Для дослідження даного питання необхідно створити вебзастосунок на основі обраного архітектурного рішення. І оскільки було обрано створити веб-афішу для дослідження даної проблеми, розглянемо що з себе представляє процес створення вебсайту. Процес створення вебсайту або вебзастосунка називається веброзробкою.

Власне процес веброзробки складається с декількох трудомістких підпроцесів, а саме:

- вебдизайн,
- верстка сторінок,
- розробка архітектури застосунку,
- програмування для веб на стороні клієнта та сервера,
- робота з базами даних,
- конфігурування вебсервера.

В даній роботі сконцентруємо увагу саме на пункті розробки архітектури веб-застосунку.

1.2 Аналіз архітектурних підходів при розробці програмного забезпечення

Розглянемо можливі підходи до проектування архітектури веб-застосунків. Серед найвідоміших виділимо монолітну архітектуру, сервіс-орієнтовану архітектуру та мікросервісну архітектуру. Кожна з них має свої переваги та недоліки які ми далі розглянемо.

1.2.1 Монолітна архітектура

Моноліт - давнє слово, що відноситься до величезного єдиного каменю. Хоча цей термін широко використовується сьогодні, зображення залишається однаковим у всіх сферах. В інженерії програмного забезпечення монолітний об’єкт відноситься до єдиної неподільної одиниці. Концепція монолітного програмного забезпечення полягає в різних компонентах програми, об'єднаних в єдину програму на одній платформі[1]. Зазвичай монолітний додаток складається з бази даних, користувацького інтерфейсу на сторони клієнта та серверного додатку. Всі частини програмного забезпечення уніфіковані, а всі його функції управляються в одному місці.

Давайте детально розглянемо структуру монолітного програмного забезпечення. Монолітна архітектура зручна для роботи невеликих команд, саме тому багато стартапів вибирають такий підхід під час створення програми. Компоненти монолітного програмного забезпечення взаємопов'язані та взаємозалежні, що допомагає програмному забезпеченню бути самостійним. Ця архітектура є традиційним рішенням для створення додатків, але деякі розробники вважають її застарілою. На рисунку 1.1 наведено діаграму структури монолітного вебзастосунку.

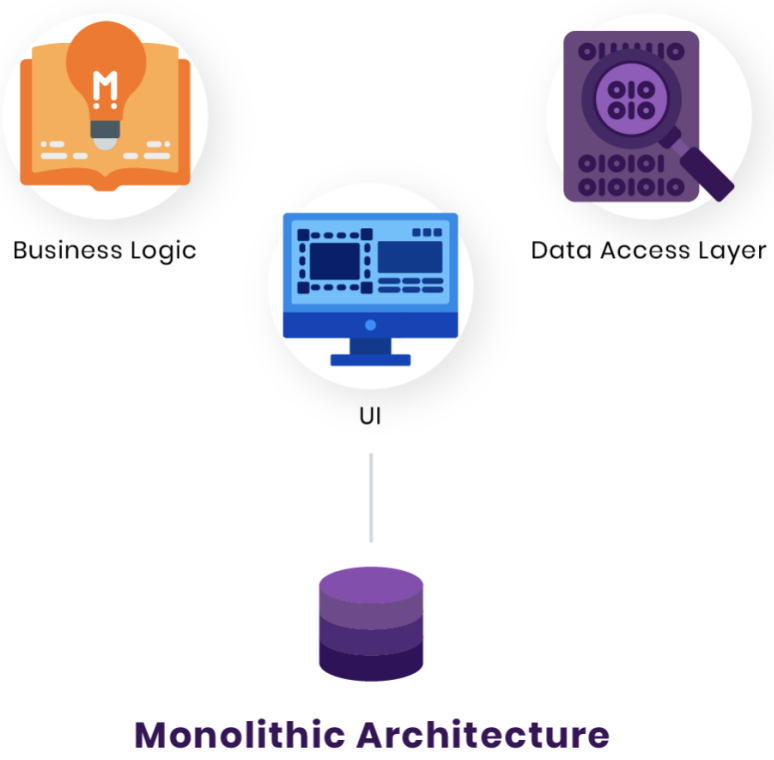


Рисунок 1.1 – Структура монолітного вебзастосунку

Розглянемо детальніше переваги монолітної архітектури[1].

– Проста розробка та процес розгортання.

Існує безліч інструментів, які можна інтегрувати для полегшення розробки монолітного застосунка. Крім того, всі дії виконуються з одним проектом, що передбачає більш легке розгортання додатку на сервері. Маючи монолітне ядро, розробникам не потрібно розгортати зміни або оновлення окремо, оскільки вони можуть це робити відразу і економити багато часу.

– Менше кросбраузерних проблем.

Більшість застосунків стикаються з проблемами кросбраузерності. Монолітні додатки вирішують ці проблеми набагато простіше за рахунок

їх єдиної кодової бази. Простіше вирішувати ці проблеми, коли все працює в одному додатку.

– Простота в масштабуванні.

Монолітні застосунки прості в масштабуванні по горизонталі шляхом запуску кількох копій для балансування навантаження.

На ранніх стадіях проекту монолітні веб-застосунки працюють добре, і в основному більшість великих і успішних додатків, які існують сьогодні, почалися як моноліт.

Але існує достатня кількість недоліків монолітної архітектури, котрі наведені нижче.

– Кодова база з часом стає громіздкою.

З часом більшість продуктів розвивається і збільшується за обсягом, і їх структура стає розмитою. Кодова база починає виглядати дійсно великою і її стає важко розуміти та змінювати, особливо для нових розробників. Також стає важче знайходити побічні ефекти та залежності. Зі зростанням проекту, якість кодової бази знижується, і інтегроване середовище розробки (IDE) перевантажується.

– Складність в інтеграції нових технологій.

Якщо потрібно додати якусь нову технологію до свого додатка, розробники можуть зіткнутися з перешкодами в адаптації коду. Додавання нової технології означає значні зміни в кодовій базі усієї програми, що дорого і забирає багато часу.

– Обмежена гнучкість.

У монолітних додатках кожне невелике оновлення вимагає повного перерозгортання додатку. Таким чином, всі розробники повинні чекати, поки це буде зроблено. Коли над одним проектом працює кілька команд, це може стати вагомою проблемою.

Монолітні програми також можуть бути важко масштабованими, коли різні модулі мають конфліктні вимоги до ресурсів.

Іншою проблемою з монолітними додатками є надійність. Помилка в будь-якому модулі (наприклад, витік пам'яті) може потенційно збити весь процес роботи додатку. Більше того, оскільки всі екземпляри програми ідентичні, ця помилка вплине на доступність всієї програми.

Монолітна модель не застаріла і вона все ще чудово працює в деяких випадках. Деякі гігантські компанії, як Etsy, залишаються монолітними, незважаючи на популярність мікросервісів сьогодні. Монолітна архітектура програмного забезпечення може бути корисною, якщо команда розробників знаходиться на стадії заснування, створюється новий продукт і у розробників немає досвіду роботи з мікросервісами. Моноліт ідеально підходить для стартапів, яким потрібно якомога швидше запуститися. Однак певні питання, згадані вище, роблять моноліти не кращим варіантом для сучасного застосунку який з часом буде рости та включати в себе нові технології.

1.2.2 Сервісно-орієнтована архітектура

Природним переходом від монолітної архітектури було використання сервісно-орієнтованої архітектури (SOA). Використовуючи такий підхід, додаток розбивається на більш дрібні модулі. Тоді всі служби працюють із шаром агрегації, який можна назвати шиною (рисунок 1.2).

SOA має дві основні ролі: постачальник послуг та споживач послуг. Обидві ці ролі може грати програмний агент. Концепція SOA полягає в наступному: додаток може бути спроектовано та побудовано таким чином, що його модулі інтегруються без жодних проблем і можуть бути легко використані[2].

Рисунок 1.2 – Структура SOA застосунку

Щодо плюсів та мінусів сервісно-орієнтованої архітектури[2]. Серед переваг розглянемо наведені нижче.

– Повторне використання сервісів.

Завдяки самодостатньому характеру функціональних компонентів у сервісно-орієнтованих додатках ці компоненти можна повторно використовувати в декількох додатках, не впливаючи на інші сервіси.

– Кращі можливості по підтримці додатку.

Оскільки кожна служба програмного забезпечення є незалежною одиницею, її легко оновлювати та підтримувати, не завдаючи шкоди іншим службам. Наприклад, великими корпоративними програмами керувати легше, коли вони розбиті на сервіси.

– Краща надійність.

Служби простіше налагодити і перевірити, ніж величезні фрагменти коду у монолітному підході. Це, у свою чергу, робить продукти на основі SOA більш надійними.

– Паралельна розробка.

Оскільки сервісно-орієнтована архітектура, складається з шарів, вона виступає за паралелізм у процесі розвитку. Незалежні служби можуть розвиватися паралельно і одночасно комплектуватися.

Але тим не менш існує і достатня кількість мінусів у сервіс-орієнтованій архітектурі.

– Складне управління.

Основним недоліком сервісно-орієнтованої архітектури є її складність. Кожна служба повинна забезпечити одночасну доставку повідомлень. Кількість цих повідомлень одночасно може перевищувати мільйон, що робить великим завданням управління всіма службами.

– Високі інвестиційні витрати.

Розвиток SOA вимагає великих передових інвестицій у людські ресурси, технології та розвиток.

– Додаткові перенавантаження.

У SOA всі входи перевіряються до того, як одна служба взаємодіє з іншою службою. При використанні декількох сервісів це збільшує час відгуку та знижує загальну ефективність.

В цілому шар агрегації (SOA шина) є найбільшою проблемою для вирішення. Проблема полягає в додаванні оперативної логіки до шини. Оскільки цей шар стає все більшим і більшим із додаванням все більшої кількості компонентів до системи, то виникають проблеми з’єднання в середині системи.

На думку експертів, ще одна найбільша проблема виникла у формі поводження з помилками. З цією архітектурою система отримує велику кількість запитів за умовну одиницю часу без перерви і не має змогу у разі

виникнення помилки обробити її належним чином, що надзвичайно ускладнює процес пошуку помилок та їх вирішення.

В цілому SAO так і не здобули своєї популярності саме через описані недоліки. Люди почали дивитися на монолітні застосунки або рухалися до мікросервісної архітектури.

1.2.3 Мікросервісна архітектура

То що ж таке мікросервіси? Офіційного визначення немає, але суть наступна. Мікросервіси представляють архітектурний стиль, в якому складні додатки створені як сукупність маленьких, легких, самодостатніх, незалежних, нетісно зв’язаних сервісів, кожен з яких відповідальний за конкретний процес (рисунок 1.3). Такий стиль протиставляється монолітному стилю, згідно з яким додатки будуються як єдине ціле[3].

Мікросервіси співпрацюють один з одним на основі потреби виконання певної дії. Вони «спілкуються» через API, для яких не має значення мова програмування.

Цей підхід нагадує команду, кожен учасник якої сконцентрований на конкретній задачі. Йому дають відповідальність, свободу і довіру щодо виконання своєї ділянки роботи найкращим чином. Зазвичай це дуже продуктивно!

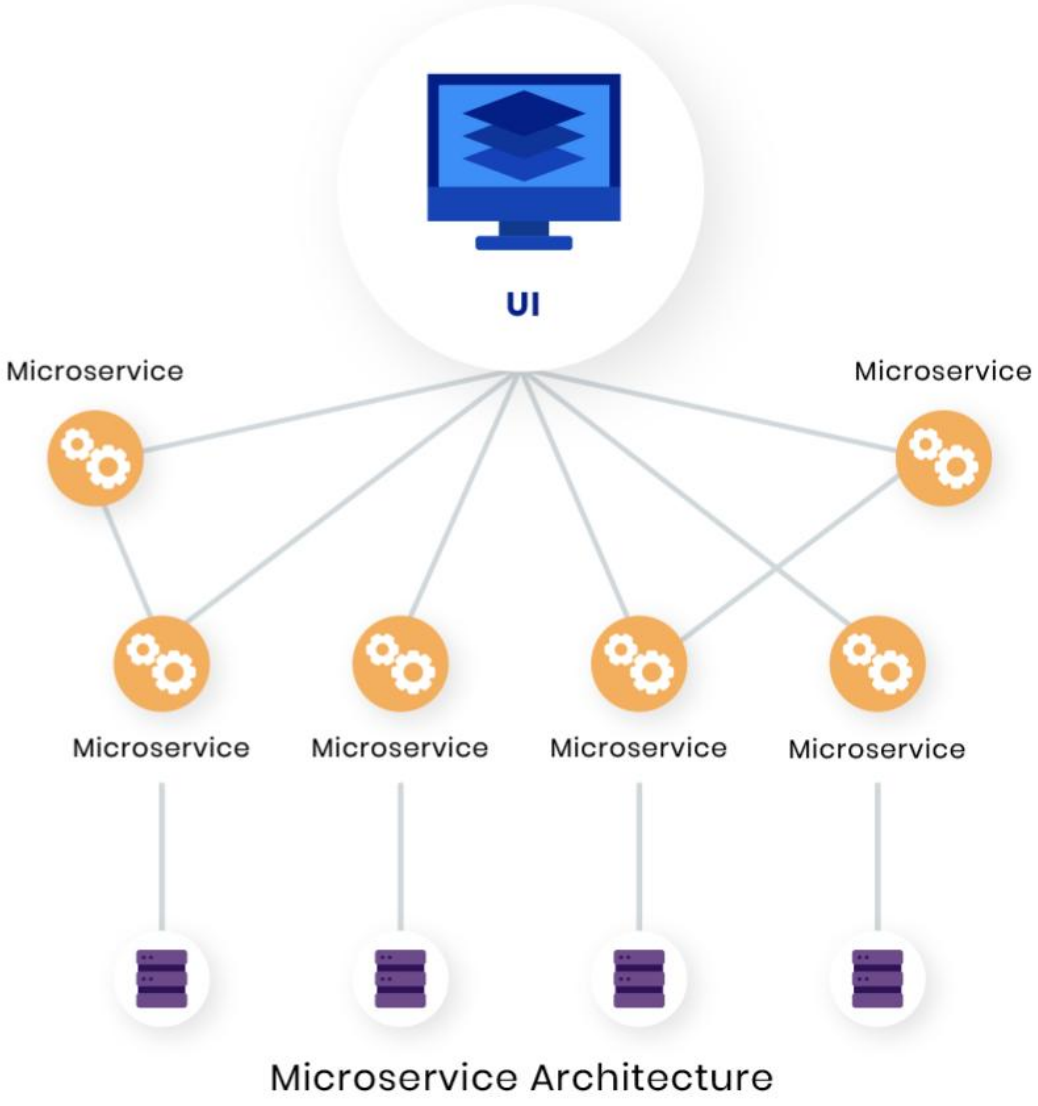


Рисунок 1.3 – Структура мікросервісної архітектури

Можна застосовувати мікросервісну архітектуру в побудові додатку з нуля. Однак, можна і монолітний додаток розбити на сервіси. Чому це варто робити? Правильно і професійно створені мікросервіси дають безліч переваг. Давайте зупинимось на деяких з них[3].

– Швидке внесення змін та легке, безпроблемне розгортання.

Кожен мікросервіс розгортається окремо. Тож якщо ви змінюєте щось в одному з них, ви можете розгорнути ці зміни, не чіпаючи інших

мікросервісів, які можуть продовжувати працювати. Можна вносити будь-які зміни настільки часто, наскільки потрібно, щоб додаток завжди відповідав вашим потребам. В монолітній архітектурі все інакше — будь-яка зміна потребує розгортання цілої складної системи.

– Повністю модернізовані додатки без зусиль.

Будь-який мікросервіс у системі можна замінити. Його можна переписати з нуля в межах прийнятного часу та бюджету без необхідності перебудовувати всю систему. Сучасний додаток, який легко розширити і переробити, та який завжди «приймає форму» всіх ваших нових ідей — це просто мрія!

Якщо ж ідеться про старий монолітний додаток, його також можна модернізувати, уникнувши «великої перебудови», якщо перевести його в «філософію» мікросервісів. Поступово відокремлювати части системи, не «зносячи» її всю, і конвертувати в мікросервіси. З часом можна отримати абсолютно новий додаток.

– Потенційно легші для розуміння, підтримки і тестування.

Мікросервіси зазвичай невеличкі за обсягом коду. Завдяки цьому, командам розробників легше їх розуміти і підтримувати — звісно, якщо мікросервіси створені правильно. Їх також легше повністю покривати автотестами.

В будь-якому випадку, тестувальникам потрібно перевіряти якусь лише невелику ділянку, тож процес іде швидше, зворотній зв’язок отримується також швидше, і команда почувається впевненіше.

– Помилка в одному мікросервісі не підірве роботу усієї системи.

Неполадки у мікросервісі не повинні зламати весь додаток. Швидше за все, вони не вплинуть суттєво на роботу додатку, особливо великого.

– Мікросервіси на різних мовах і платформах можуть працювати разом.

Як хороші учасники команди, мікросервіси можуть бути абсолютно різними і при цьому проявляти свої найкращі якості в командній роботі. Їм лише потрібно знати, як комунікувати одне з одним. У випадку мікросервісів комунікація відбувається найчастіше за допомогою HTTP, а також — сервісної шини, бінарного протоколу тощо.

Можливість використовувати різні мови програмування та інструменти — це індивідуальний підхід до побудови сервісів. Він дозволяє обирати ідеальний інструмент для конкретної роботи, а також долучати нові технології, коли забажаєте.

– Різні команди можуть працювати з різними сервісами.

З вищесказаного випливає, що у вас може бути декілька команд, які працюватимуть над різними частинками додатку. І кожна команда може проявити себе найкраще у тому, на чому спеціалізується.

Щодо мінусів мікросервісів, так головним мінусом даного архітектурного рішення є складність. Підтримувати роботу великої кількості компонентів, їх незалежного розгортання, логування, тощо є надзвичайно тяжким процесом.

Підсумуємо переваги та недоліки описаних архітектурних рішень в таблиці 1.1.

Таблиця 1.1 – Порівняння архітектурних підходів до розробки вебзастосунків

	Мікросервіси	SOA	Монолітна архітектура
Дизайн	Сервіси побудовані як невеликі одиниці, виражені формально за допомогою API, орієнтованого на бізнес.	Послуги можуть бути різного розміру від невеликих додатків до дуже великих корпоративних сервісів, включаючи досить великі бізнес компоненти.	Монолітні програми виростають до величезних розмірів, і виникають ситуації, коли зрозуміти всю програму складно.
Зручність використання	Служби, що піддаються стандартному протоколу, наприклад, RESTful API, і споживаються/повторно використовуються іншими службами та програмами.	Сервіси, що піддаються стандартному протоколу, такому як SOAP та споживаються/повторно використовуються іншими службами.	Повторне використання монолітних додатків дуже обмежене.
Масштабованість	Сервіси існують незалежні компоненти для розгортання і наявна можливість окремо масштабувати кожен сервіс.	Залежності між сервісами та багаторазовими компонентами можуть спричинити проблеми з масштабування.	Масштабування монолітних додатків є задачею доволі тяжкою.

Продовження таблиці 1.1

Гнучкість	Малі незалежні сервіси полегшують управління циклом розробки, тим самим надаючи високу гнучкість.	Висока зв’язаність між компонентами обмежує можливості менеджменту системи.	Для монолітних застосунків досить тяжко досягнути гнучкості у розгортанні.
Розробка	Розробка сервісів роздільно дозволяє командам використовувати необхідні їм фреймворки для розробки.	Багаторазові компоненти та стандартні практики допомагають розробникам у процесі розробки.	Монолітні застосунки розробляються за допомогою єдиного стека технологій, таких як (JEE, .NET, тощо). Це в свою чергу певним чином обмежує розробників у процесі розробки та з вибором найкращої технології для розробки.

На основі цього можливо зробити висновок, що монолітні програми складаються з взаємозалежних, неподільних одиниць і відрізняються дуже низькою швидкістю розвитку. SOA розбивається на більш дрібні, помірно поєднані сервіси та все ще мають повільний розвиток, а мікросервіси - це дуже малі, нещільно пов'язані незалежні сервіси, що мають швидкий постійний розвиток.

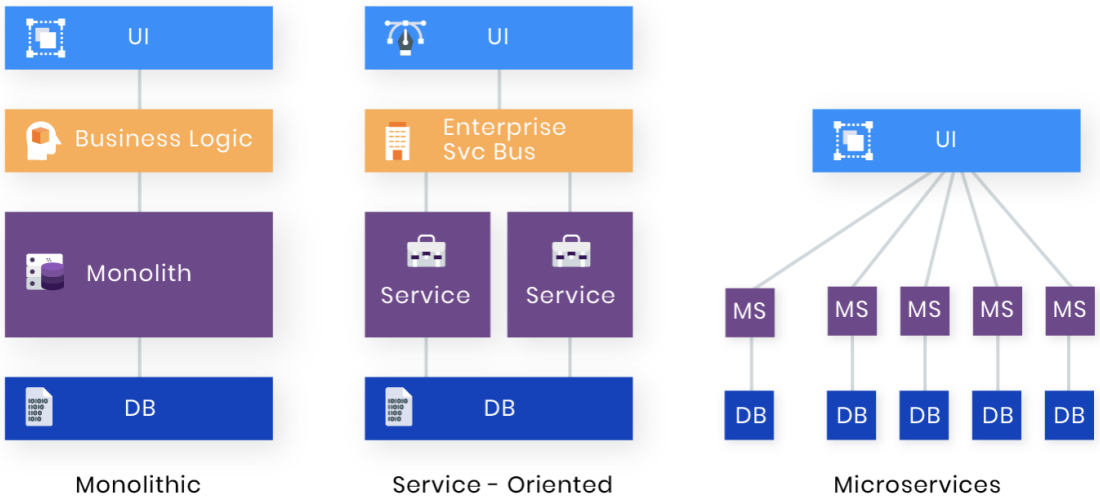


Рисунок 1.4 - Порівняння архітектурних рішень Monolith vs SOA vs MSA

1.3 Висновки до розділу

Існує достатня кількість архітектурних підходів до розробки веб-застосунків. Було порівняно три з них, це монолітні вебзастосунки, сервісно-орієнтовані застосунки та застосунки побудовані на основі мікросервісної архітектури. Був проведений детальний аналіз усіх переваг та недоліків кожного архітектурного підходу та було наведено схематичні діаграми їх структури. І хоча кожне з розглянутих архітектурних рішень має свої переваги та недоліки, але під сучасні реалії розробки ПЗ, саме мікросервісна архітектура є найбільш привабливою. Тим не менш вагомим недоліком мікросервісної архітектури являється складність процесу розробки такої системи, тому необхідно розглянути ближче особливості даного архітектурного рішення та виокремити засоби, які допоможуть вирішити проблеми мікросервісної архітектури.

2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ

2.1 Загальний опис мікросервісної архітектури

Скажемо, Ви розробляєте корпоративну програму на сервері. Вона повинна підтримувати безліч різних клієнтів, включаючи настільні браузери, мобільні браузери та гібридні мобільні додатки. Додаток має також надавати API для використання сторонніми ресурсами. Він також має інтегруватися з іншими програмами через веб-сервіси чи брокери повідомлень. Додаток також має обробляти запити (HTTP-запити та повідомлення), виконуючи бізнес-логіку, надавати доступ до бази даних, дозволяти обмін повідомленнями з іншими системами і повертати відповіді в форматах HTML/JSON/XML. Саме у таких випадках можливо вдало використати мікросервісну архітектуру.

Таке архітектурне рішення надає масу переваг та дозволяє вирішити описані вище проблеми. Тож використання мікросервісної архітектури надає такі переваги:

– з’являється можливість використовувати безперервну доставку та розгортання великих, складних додатків;

– кожен сервіс є відносно малих розмірів, а відповідно легко розуміється та піддається змінам та масштабуванню;

– Невеликий розмір сервісів дозволяє їх швидко та зручно тестувати;

– спрощення розгортання додатку на сервері та можливість розгортати сервіси на сервері не зупиняючи роботу усього додатку;

– мікросервісна архітектура дозволяє організувати зусилля з розробки навколо декількох автономних команд. Кожна команда є власником і відповідає за один або кілька сервісів. Кожна команда може розробляти, тестувати, розгортати та масштабувати свої

послуги незалежно від усіх інших команд. Це все надає неабияку гнучкість при розробці;

– малий розмір кожного мікросервісу має ще більше переваг;

– сервіси легше зрозуміти розробникам, що пришвидшує процес розробки;

– IDE працює швидше, що робить розробників продуктивнішими;

– запуск кожного мікросервісу окремо є доволі швидким процесом, що доволі сильно пришвидшує і процес розробки и і процес розгортання додатку на сервері;

– поліпшена ізоляція помилок. Наприклад, якщо в одному сервісі є витік пам'яті, це вплине лише на цей сервіс. Інші сервіси продовжуватимуть обробляти запити. Для порівняння, один зламаний компонент монолітної архітектури може збити всю систему;

– дозволяє гнучко обирати технології для розробки. При розробці нового сервісу ви можете вибрати новий стек технологій. Так само, вносячи істотні зміни до існуючого сервісу, ви можете переписати його за допомогою нового стека технологій.

Але потрібно розуміти, що все ж усі ці переваги мають певну ціну, а саме складність в реалізації та підтримці системи. Тим не менш існує досить велика кількість шаблонів ціллю яких є спрощення системи та процесу її розробки та підтримки (рисунок 2.1).

2.2 Шаблони мікросервісної архітектури

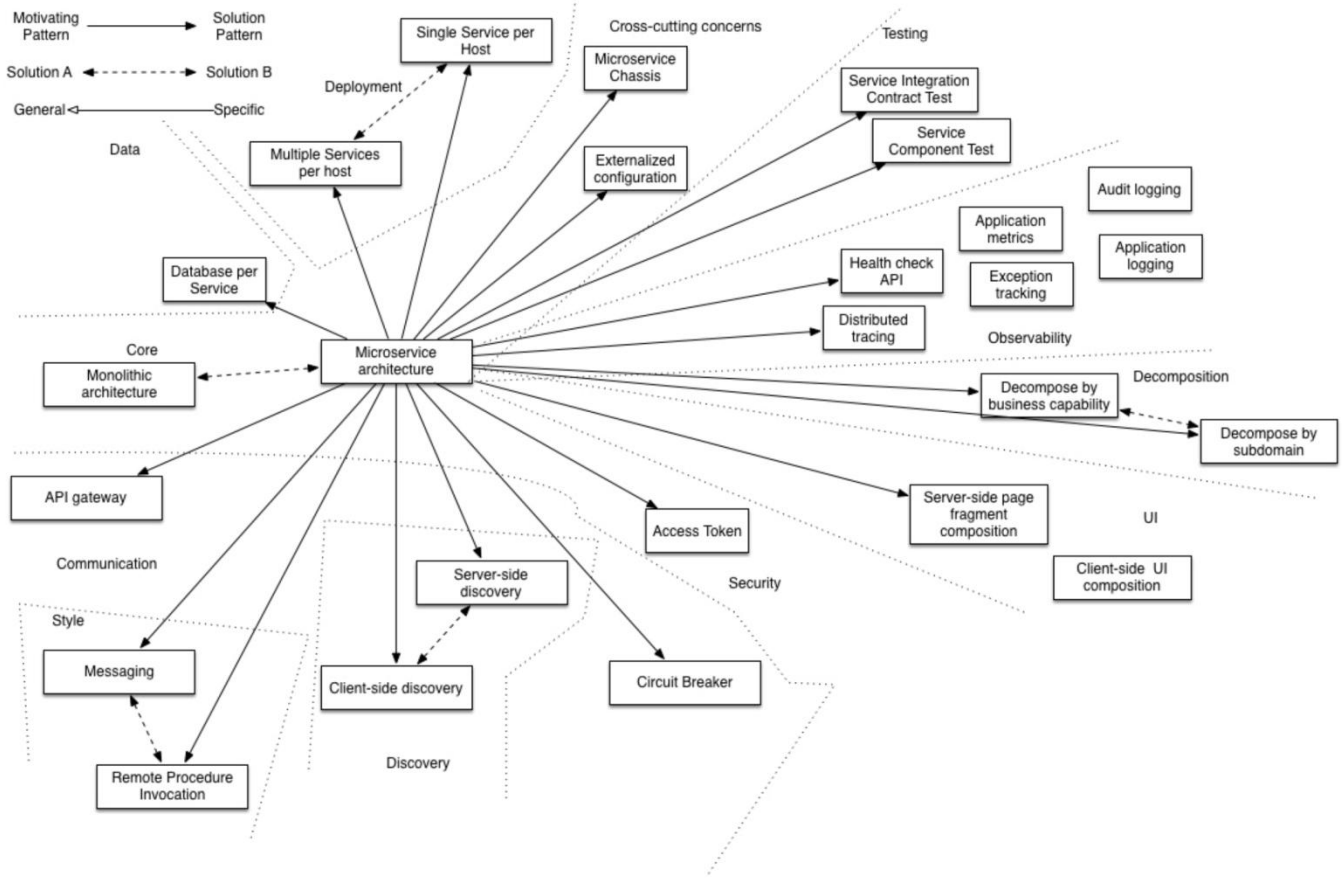


Рисунок 2.1 – Шаблони мікросервісної архітектури

2.2.1 Шаблон «Decompose by business capability»

Скажемо розроблювана нами система ставить перед нами такі задачі:

– архітектура застосунку має бути стабільна;

– сервіси мають бути згуртовані. Сервіс має реалізовувати невелику кількість сильно пов’язаних функцій;

– сервіси мають бути розроблені згідно загальному принципу закриття - речі, які змінюються разом, повинні бути упаковані разом - щоб гарантувати, що кожна зміна стосується лише одного сервісу;

– служби повинні бути слабко пов'язані - кожний сервіс як API, який інкапсулює його реалізацію, де реалізацію можна змінити, не впливаючи на клієнтів;

– сервіс необхідно просто тестувати;

– кожен сервіс має бути досить малим щоб його могла розробляти команда в 6 -10 розробників;

– кожна команда, яка володіє одним або кількома сервісами, повинна бути автономною. Команда повинна вміти розробляти та розгортати свої сервіси при мінімальній співпраці з іншими командами.

Рішенням даних завдань є розбиття системи на сервіси згідно бізнес можливостей. Бізнес можливість це концепція з моделювання архітектури бізнесу. Власне це щось, що робить бізнес щоб отримати результат. Частіше за все бізнес можливості спираються на бізнес об’єкти, для прикладу менеджмент замовлень відповідає за замовлення, а менеджмент користувачів відповідає за користувачів. Наведемо приклад, скажемо ми маємо такі бізнес можливості:

– менеджмент продукції;

– менеджмент запасами;

– менеджмент замовлень;

– менеджмент доставки.

Тоді відповідна мікросервісна архітектура буде мати наступне розбиття на сервіси на основі бізнес можливостей (рисунок 2.2).

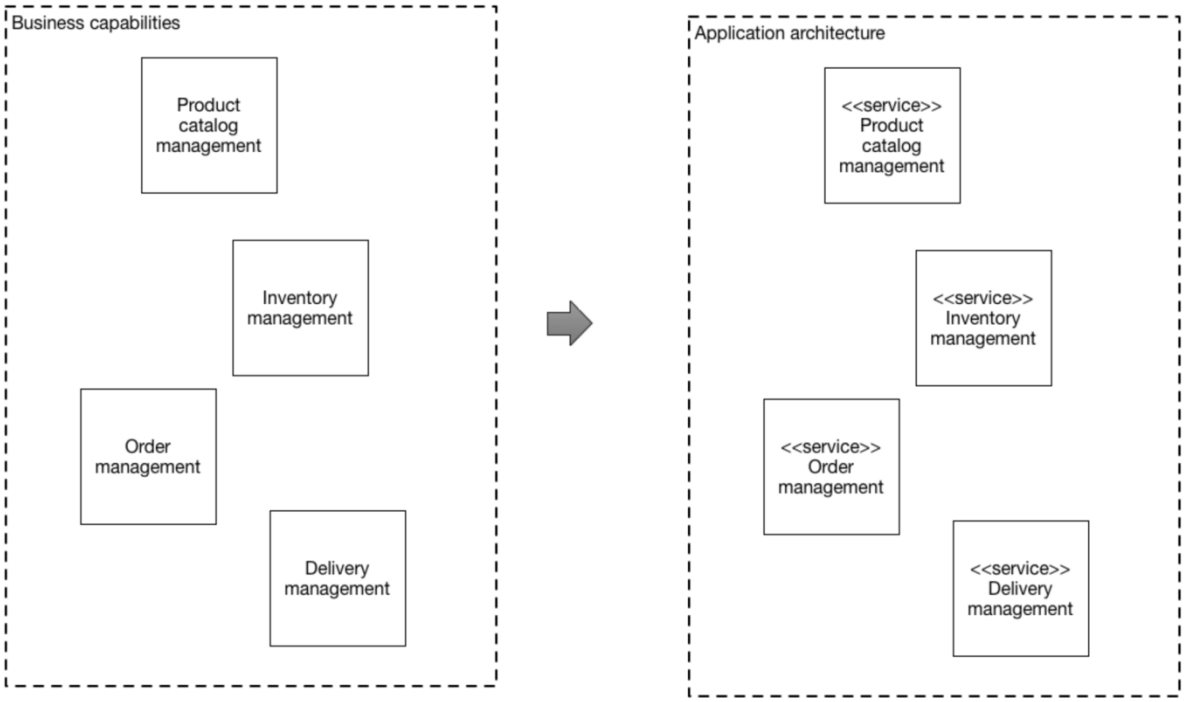


Рисунок 2.2 – Приклад розбиття системи по бізнес можливостям

Даний шаблон має наступні переваги:

– стабільність архітектури, оскільки бізнес можливості частіше за все є стабільними;

– команди розробників мультифункціональні, автономні та організовані навколо бізнес вимог, а не навколо технічних аспектів;

– сервіси слабко пов’язані між собою;

– сервіси згуртований, тобто містить тісно пов’язані між собою функції.

2.2.2 Шаблон «Database per service»

Даний шаблон вирішує який принцип потрібно використовувати при роботі з БД в застосунку на основі мікросервісів. Також він вирішує наступні проблеми:

– сервіси повинні бути слабко пов'язані, щоб їх можна було самостійно розробляти, розгортати та масштабувати;

– деякі бізнес транзакції мають модифікувати сутності з різних сервісів;

– деякі запити мають мати змогу діставати дані, що належать іншим сервісам;

– бази даних мають мати можливість бути скопійованими та мігрованими;

– різні сервіси мають мати змогу використовувати різні бази даних. Для деяких випадків більш підходять реляційні БД, для деяких краще використовувати NoSQL, такі як MongoDB, які краще підходять для зберігання складних, неструктурованих даних.

Головною ідеєю є зберігати дані кожного мікросервісу приватними для цього мікросервісу та доступними лише через його API. Транзакції сервісу включають лише роботу з його базою даних.

На наступному рисунку представлена ідея даного шаблону.

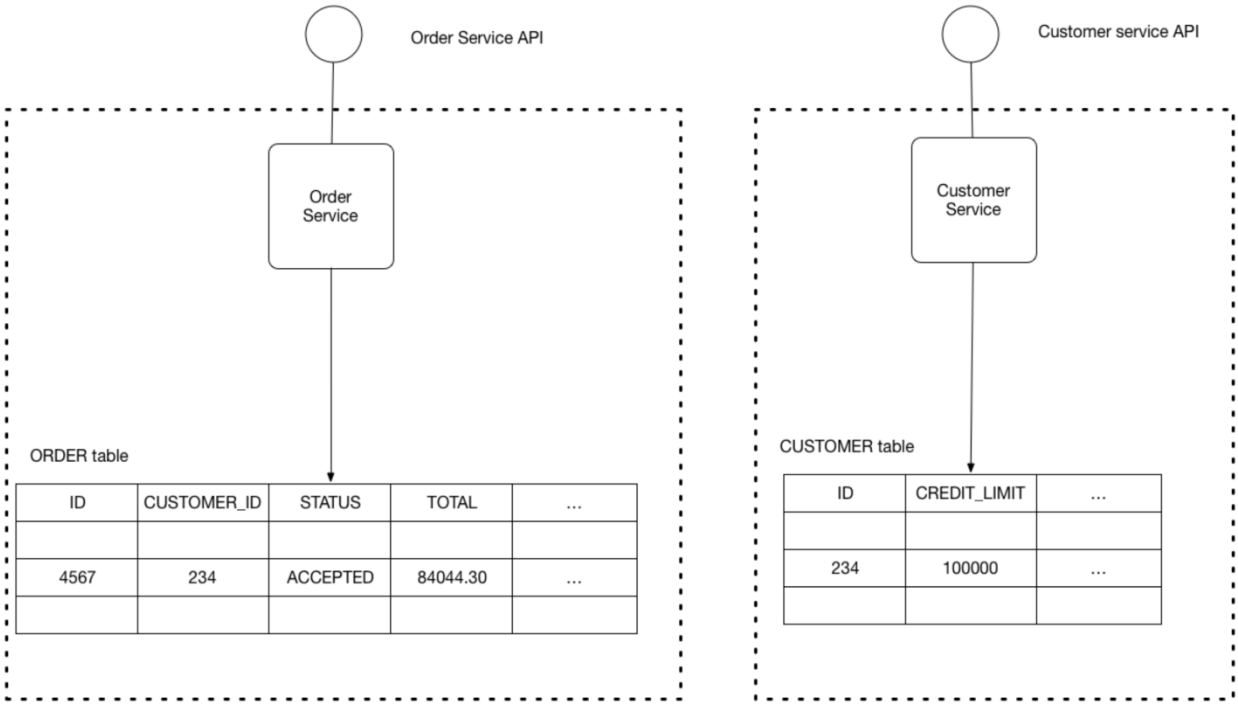


Рисунок 2.3 – Шаблон «Database per service»

База даних сервісу являється частиною сервісу і доступ до неї можливо отримати лише через API самого сервісу.

Існує декілька можливостей зберігати дані, що відносяться до бізнес логіки певного сервісу. Власне не обов’язково створювати окрему базу даних для кожного сервісу. Скажемо, якщо використовується реляційна база даних, то можливі наступні варіанти:

– окрема таблиця для окремого сервісу – кожен сервіс має свій набір таблиць до який можна отримати доступ лише за допомогою API даного сервісу, але при цьому саму БД використовують і інші сервіси;

– окрема схема для окремого сервісу – кожен сервіс має свої схеми до яких можна отримати доступ лише за допомогою API даного сервісу;

– окрема БД для окремого сервісу – кожен сервіс має свою БД.

Підсумовуючи, можна сказати, що даний шаблон надає нам наступні переваги:

– допомагає впевнитись, що сервіси є слабко зв’язаними. Внесення змін до однієї БД не впливає на інші;

– кожен сервіс має можливість використовувати свій тип БД, що додає гнучкості при розробці.

2.2.3 Шаблон «API Gateway»

Даний шаблон, власне, є одним з ключових при розробці системи на основі мікросервісів.

Використання API Gateway добре себе оправдовує коли розробляються та створюються великі або складні програми на основі мікросервісної архітектури з декількома клієнтськими частинами. В цілому це сервіс, який забезпечує єдину точку входу для певних груп мікросервісів. Цей шаблон схожий на модель "Фасад" з об'єктно-орієнтованого дизайну, але в цьому випадку це частина розподіленої системи. Шаблон API Gateway іноді також відомий як "бекенд для інтерфейсу" (BFF)[4].

Тому API Gateway знаходиться між клієнтською частиною та мікросервісами. Він виступає як зворотний проксі, пов’язуючи запити від клієнтів до служб. Він також може забезпечити додаткові наскрізні функції, такі як автентифікація, обробка SSL та кешування. На рисунку 2.4 зображено схематично структуру даного шаблону.

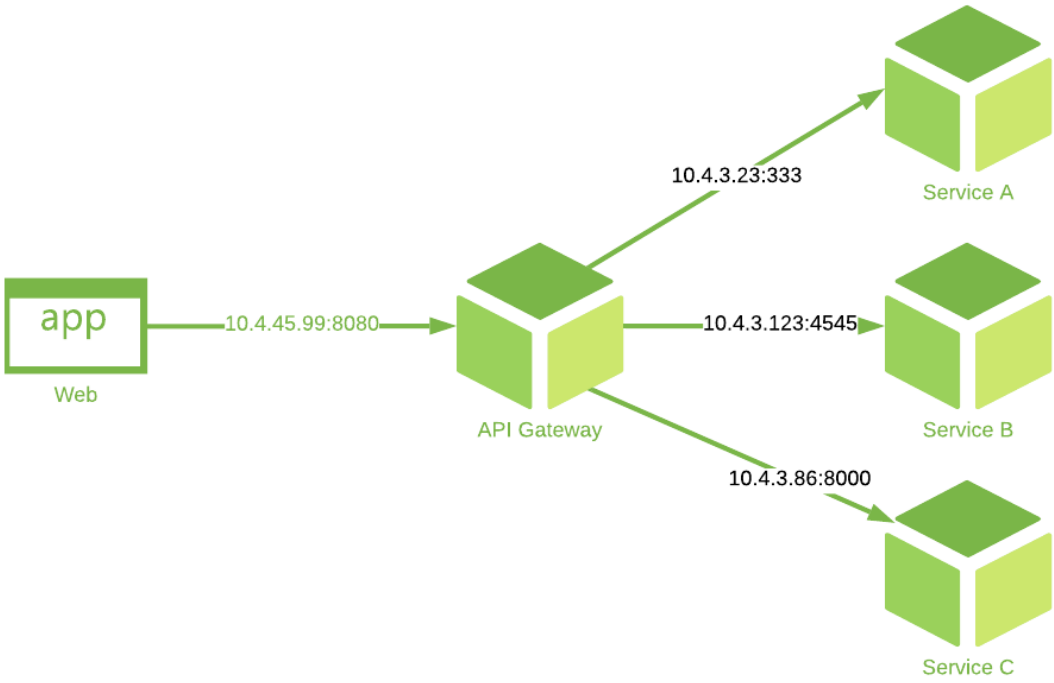


Рисунок 2.4 – API Gateway

Виникненню даного шаблону сприяли виникнення таких проблем при розробці систем на основі мікросервісів.

– Роздільність API, що надаються мікросервісами, частіше за все відрізняються від тих, що необхідні клієнту. Мікросервіси зазвичай надають зернисті API, що означає, що клієнтам потрібно взаємодіяти з декількома службами. Наприклад, коли клієнт хоче отримати дані про користувача, йому можуть знадобитись інші сервіси окрім сервісу роботи з користувачами, якщо скажем для клієнтської частини необхідні і дані скільки цей користувач зробив покупок, і на які події підписаний, тощо.

– Різні клієнти потребують різних даних. Наприклад, версія даних які необхідні браузерній версії додатку, може відрізнятись від версії даних які необхідні мобільному застосунку.

– Продуктивність мережі відрізняється для різних типів клієнтів. Наприклад, мобільна мережа, як правило, набагато повільніше і має значно більшу затримку, ніж немобільна мережа. І, звичайно, будь-яка WAN набагато повільніше, ніж LAN. Це означає, що гібридний мобільний клієнт використовує мережу, яка відрізняється характеристиками продуктивності, ніж локальна мережа, що використовується веб-додатком на сервері. Веб-додаток на стороні сервера може надіслати кілька запитів до сервісів сервера, не впливаючи на користувальницький досвід, коли як мобільний клієнт може зробити лише кілька.

– Кількість розгорнутих екземплярів сервісів та їх місцезнаходження (хост+порт) можуть динамічно змінюватися.

– Розмежування в сервісах може змінюватися з часом і його слід приховувати від клієнтів.

– Служби можуть використовувати різноманітний набір протоколів, деякі з яких можуть не бути зручними для Інтернету.

Реалізацією API Gateway, є єдина точка входу для всіх клієнтів. Оскільки API Gateway знаходиться між усіма запитами від клієнта до окремих сервісів, він також виступає (PEP) для запитів до сервісів. Використання централізованого PEP означає, що наскрізні проблеми, пов'язані сервісами, можуть бути вирішені в одному місці, уникаючи необхідності щоб окремі команди розробників вирішувала ці проблеми та розробляла додаткові сервіси.

2.2.4 Шаблон «Client-side service discovery»

Сервіси зазвичай виконують запити один одному. У монолітній програмі сервіси роблять запити один до одного методами на рівні мови або ж викликами на процедурному рівні. У традиційному розгортанні розподіленої системи сервіси працюють у фіксованих, добре відомих місцях (хости і порти), і тому вони можуть легко зробити запит один одному за допомогою HTTP/REST або якогось механізму RPC. Однак сучасний додаток на основі мікросервісів, як правило, працює у віртуалізованих або контейнерних середовищах, де кількість екземплярів сервісу та їх розташування динамічно змінюються.

Даний патерн виник через наступні складнощі при побудові системи на основі мікросервісів:

- кожен екземпляр сервісу надає віддалений API, такий як HTTP / REST тощо в певному місці (хост і порт);
- кількість екземплярів сервісів та їх розташування динамічно змінюються;
- віртуальним машинам і контейнерам зазвичай призначаються динамічні IP-адреси;
- кількість екземплярів сервісів може динамічно змінюватися.

Рішенням всіх цих складнощів і став даний патерн. Власне здійснюючи запит до сервісу, клієнт отримує місце розташування екземпляра сервісу шляхом запиту в SR(Service Registry), який знає місця розташування всіх екземплярів сервісу. Схематично даний шаблон зображений на рисунку 2.5.

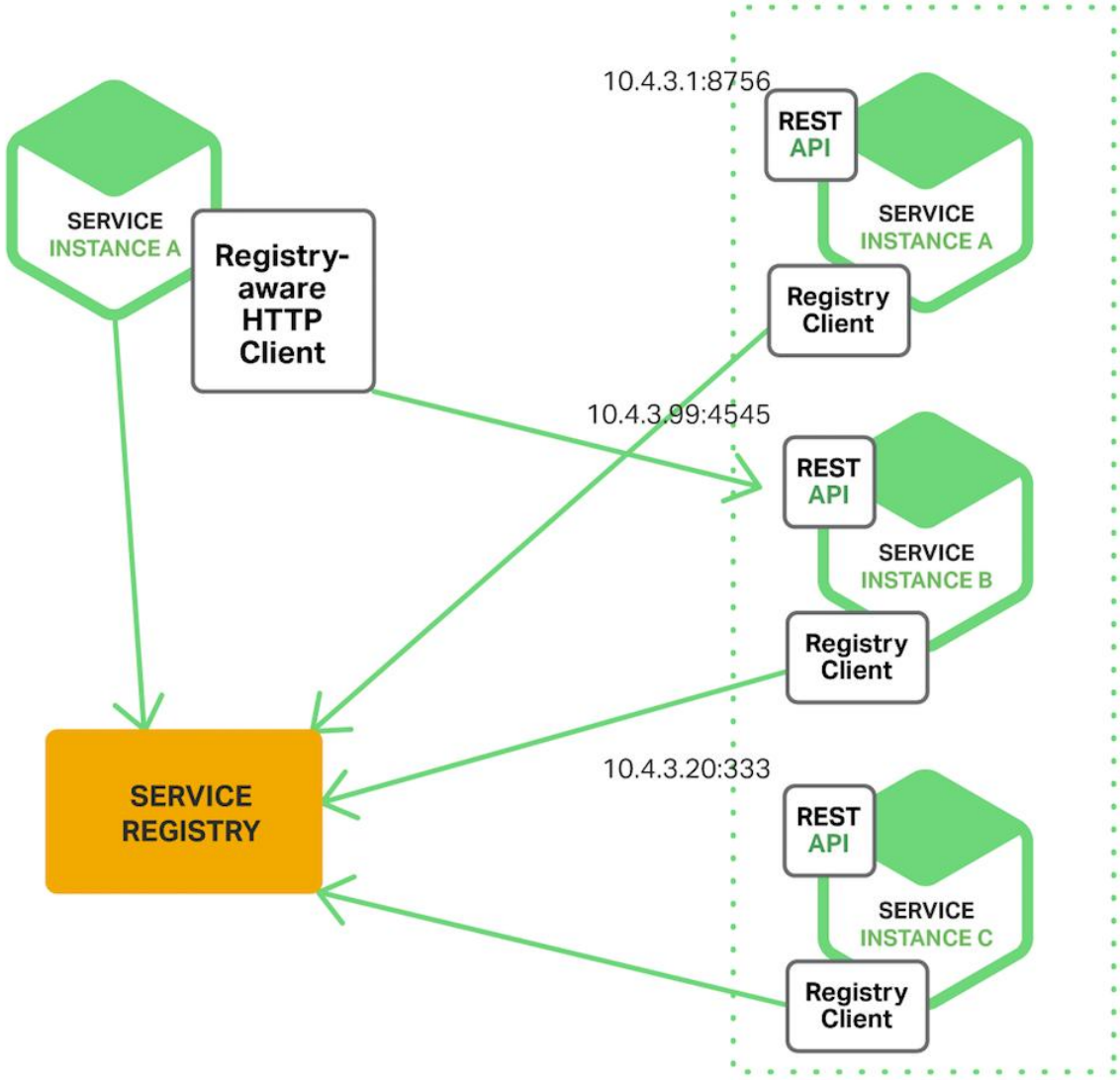


Рисунок 2.5 – Шаблон «Client-side service discovery»

2.2.5 Шаблон «Server-side service discovery»

Даний шаблон є альтернативою до описаного вище, тобто до шаблону «Client-side service discovery» і виник через ті ж самі проблеми. В чому ж різниця між Server-side service discovery та Client-side service discovery. При використанні шаблону «Server-side service discovery» здійснюючи запит до сервісу, клієнт робить запит через маршрутизатор (a.k.a Load Balancer), який працює у добре відомій місцевості.

Маршрутизатор запитує Service Registry, який може бути вбудований в маршрутизатор, і передає запит доступному екземпляру сервісу.

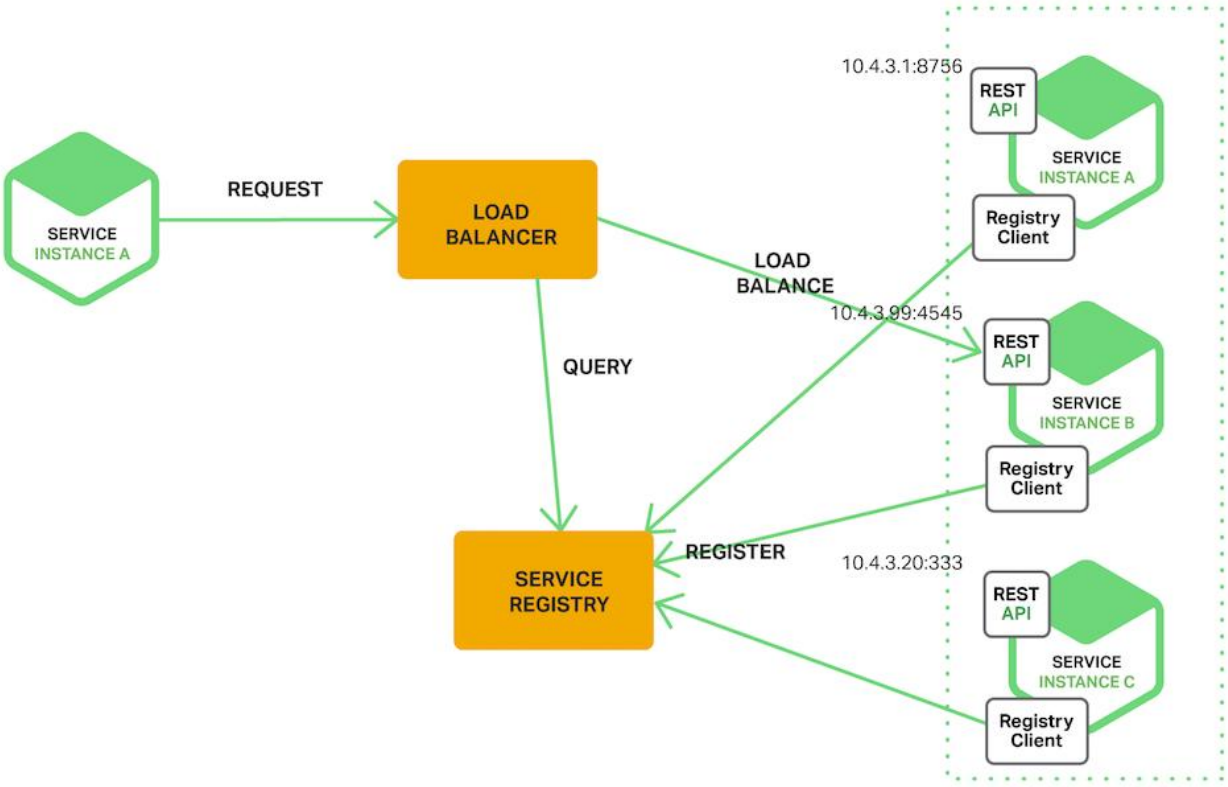


Рисунок 2.6 – Шаблон «Server-side service discovery»

2.2.6 Шаблон «Externalized configuration»

Даний шаблон відповідає на питання як дозволити сервісу працювати на різних серверах з різними налаштуваннями та мати можливість модифікувати налаштування без перерозгортання сервісів.

Шаблон «Externalized configuration» вирішує наступні виклики:

– сервіс повинний бути забезпечений даними конфігурації, які вказують, як підключитися до зовнішніх/сторонніх служб. Наприклад, розташування мережі та облікові дані бази даних;

– сервіси повинні працювати в декількох середовищах - dev, test, qa, staging, production - без модифікації та/або перекомпіляції;

– у різних середовищах різні екземпляри зовнішніх/сторонніх служб.

Головною ідеєю шаблону є екстерналізація усього додатку, включаючи облікові дані бази даних, мережеве розташування, тощо. При запуску сервіс зчитує конфігурацію із зовнішнього джерела, наприклад змінні середовища ОС, конфігураційні файли з системи GIT, тощо.

Наведемо приклад, візьмемо мову програмування Kotlin та веб фреймворк Spring. Разом вони дозволяють створювати наступні конструкції коду, що зображені на рисунку 2.7.

```
@Component
class RegistrationServiceProxy @Autowired()(restTemplate: RestTemplate)
extends RegistrationService {

  @Value("${user_registration_url}")
  var userRegistrationUrl: String = _
```

Рисунок 2.7 – Приклад використання конфігураційних змінних

Де user_registration_url це змінна, яка динамічно зчитується з сервісу конфігурації Spring, що дає можливість задати необхідні змінні для різних середовищ та ситуацій. Якщо є необхідність змінити конфігурацію для певного серверу, тощо, можливо просто змінити конфігураційні файли які зберігаються як системні файлі або файли під GIT, тощо. Сервер конфігурацій ж автоматично розпізнає зміни та обновить змінену змінну, після чого сервіси, що її використовують, отримають вже оновлене значення.

2.2.7 Шаблон «Circuit Breaker»

Контролювати велику кількість сервісів досить тяжко. Скажемо ми маємо сервіс А, що робить запит до сервісу В, а сервіс В робить запит ще до декількох сервісів і так далі. І ось один із сервісів дає збій, як тоді уникнути каскадного впливу на інші сервіси та мати змогу правильним чином обробити помилку? Цю проблему вирішує шаблон Circuit Breaker[5]. Ідея цього шаблону в тому, що клієнт сервісу повинен робити

запит до віддаленої послуги через проксі-сервер, який функціонує аналогічно електричному запобіжнику. Коли кількість послідовних відмов переступає поріг, запобіжник вимикається, і протягом тривалості періоду очікування всі спроби викликати віддалений сервіс негайно припиняються. Після закінчення часу блокування, запобіжник дозволяє виконати обмежену кількість тестових запитів. У разі успіху цих запитів запобіжник відновить нормальну роботу. В іншому випадку, якщо є збій, період очікування починається знову.

2.3 Загальна схема прецедентів

Побудуємо діаграму прецедентів (Use Case Diagram), що дозволить краще зрозуміти яку саме поведінку має реалізувати система та яку функціональність мати. Після чого буде можливим проектувати систему.

На основі вимог до системи були виокремлені наступні актори.

– Незареєстрований користувач – користувач який зайшов на веб-сайте та не автентифікувався у системі. Даний користувач має змогу ознайомитись з системою та має доступ лише до базової сторінки.

– Зареєстрований користувач – користувач, що зареєструвався у системі та відповідно має свій акаунт. Даний користувач має змогу увійти до системи, переглянути та редагувати власну інформацію.

– Адміністратор – користувач, який маж доступ до всіх частин сайту та до всієї функціональності.

Схематично дані актори зображені на рисунку 2.8.

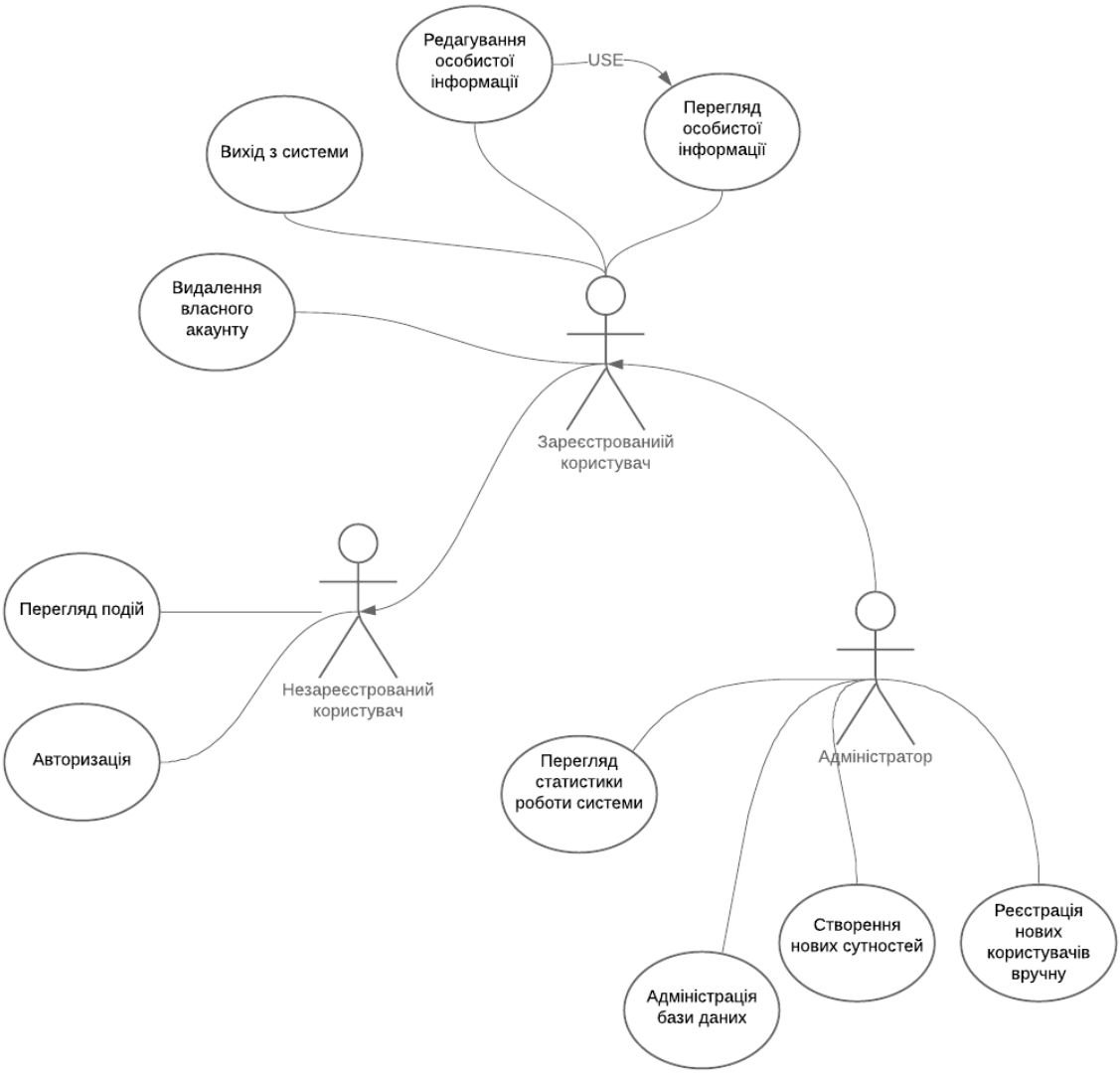


Рисунок 2.8 – Загальна схема прецедентів

На основі побудованої діаграми прецедентів ми можемо розробити сценарії прецедентів. Сценарій прецедентів – це певні дані, що описують конкретну взаємодій між користувачем та системою.

2.4 Огляд мов та веб-фреймворків

2.4.1 C#/.NET

C# є мовою з Сі-подібним синтаксисом і близькою в цьому відношенні до C++ і Java.

C# є об'єктно-орієнтованою мовою і в цьому плані багато перейняла у Java і C++. Наприклад, C# підтримує поліморфізм, успадкування, перевантаження операторів, статичну типізацію. Об'єктно-орієнтований підхід дозволяє вирішити завдання з побудови великих, але в той же час гнучких, масштабованих і розширюваних додатків. І C# продовжує активно розвиватися, і з кожною новою версією з'являється все більше цікавих функціональностей, як, наприклад, лямбда вирази, динамічне зв'язування, асинхронні методи і так далі [6].

Коли говорять C#, нерідко мають на увазі технології платформи .NET (Windows Forms, WPF, ASP.NET, Xamarin). І, навпаки, коли говорять .NET, нерідко мають на увазі C#. Однак, хоча ці поняття пов'язані, ототожнювати їх невірно. Мова C# була створена спеціально для роботи з фреймворком .NET, проте саме поняття .NET дещо ширше.

Якось Білл Гейтс сказав, що платформа .NET - це найкраще, що створила компанія Microsoft[6]. Можливо, він мав рацію. Фреймворк .NET представляє потужну платформу для створення додатків.

Основні конструктивні особливості даної платформи це[6]:

– сумісність: що дозволяє програмам, розробленим на .NET, отримати доступ до функціональних можливостей програм, розроблених поза .NET;

– загальний двигун виконання: Також відомий як загальна мова виконання, що дозволяє програмам, розробленим на .NET, обробляти загальну поведінку при використанні пам'яті, обробці винятків та безпеці;

– мовна незалежність: Загальні специфікації мовної інфраструктури (CLI) дозволяють обмінюватися типами даних між двома програмами, розробленими різними мовами;

– бібліотека базових класів: бібліотека коду для найбільш поширених функцій, що використовується програмістами, щоб уникнути повторного переписування коду;

– простота розгортання: Існують інструменти для забезпечення простоти встановлення програм без втручання у встановлені раніше програми.

.NET також дозволяє розроблювати застосунки і на основі мікросервісів. Власне для цього необхідно створити звичайний .Net веб-додаток.

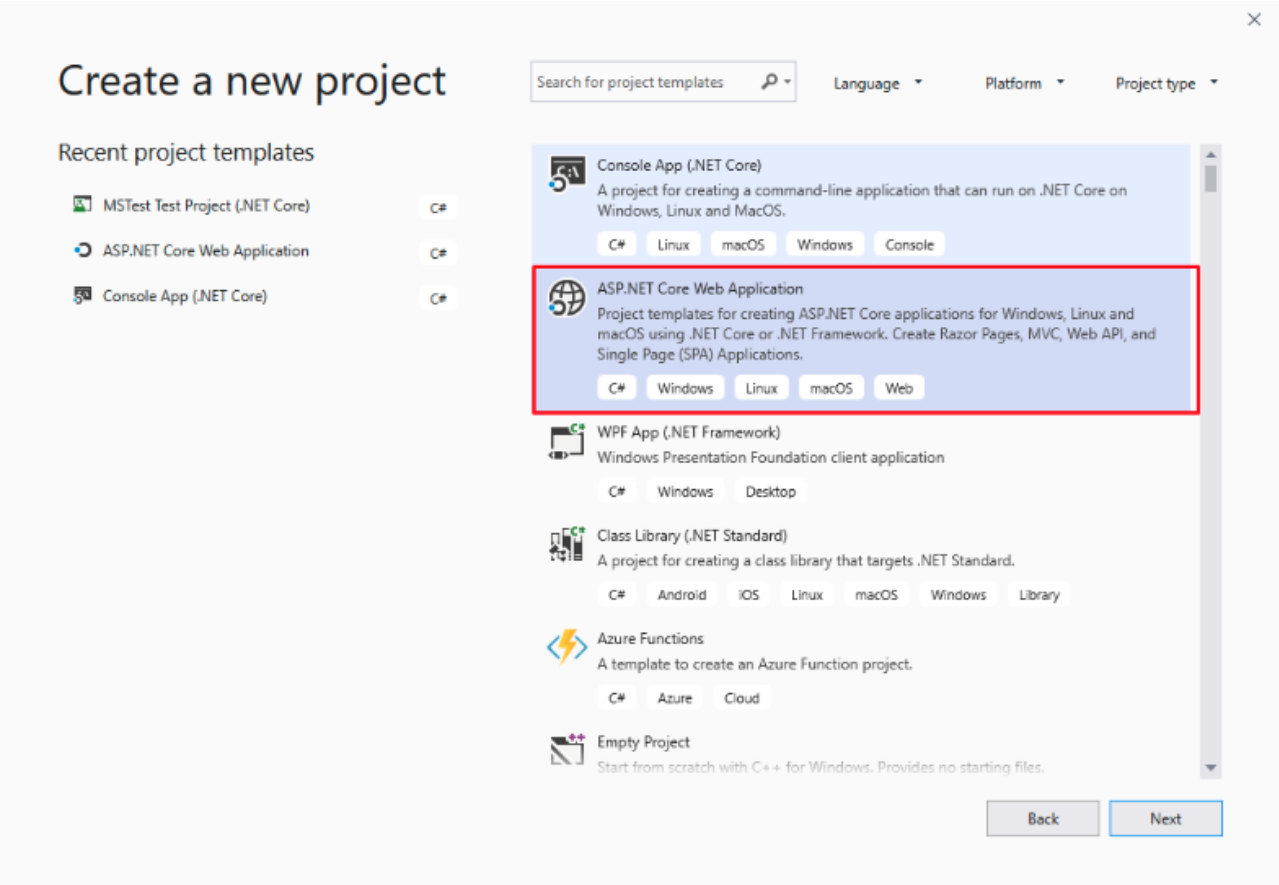


Рисунок 2.9 – Створення нового проекту на платформі .NET

Після чого необхідно обрати «API» опцію в наступному вікні та обрати підтримку Docker (рисунок 2.10).

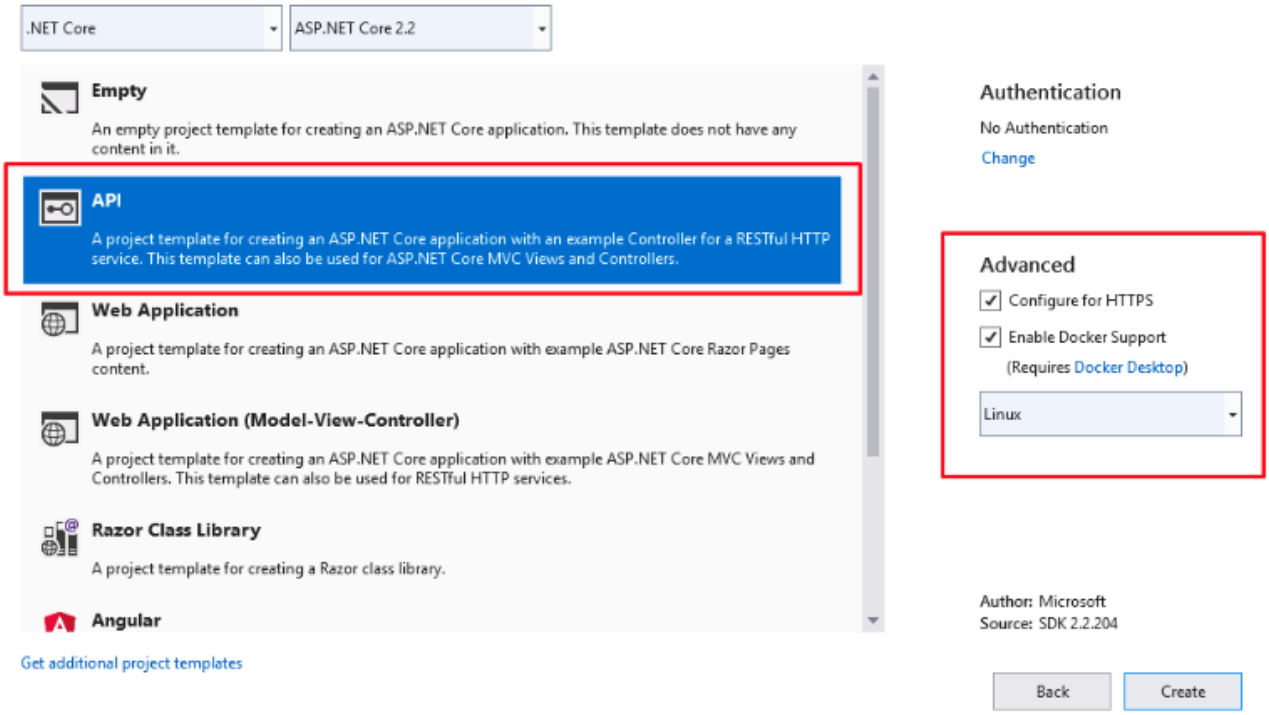


Рисунок 2.10 – Конфігурування нового проекту для платформи .NET

Далі відповідно йде процес розробки веб-сервісу та налаштування Docker. Реалізації усіх патернів для мікросервісів, наявні як велика кількість бібліотек та проектів, що можливо підключити до розроблюваного застосунку. Для прикладу реалізацією API Gateway для .NET платформи є Ocelot, що сам по собі являється відкритим проектом побудованим з використанням мови C#, що підтримує балансування навантаження, кешування, може виступати як сервер авторизації або ж я проксі сервер.

2.4.2 Java/Spring

Java - мова програмування високого рівня, спочатку розроблена Sun Microsystems та випущена в 1995 році. Java працює на різних платформах, таких як Windows, Mac OS та різних версіях UNIX[7]. До списку ключових переваг мови можна включити[8]:

– об’єктно-орієнтована мова - У Java все є Об'єктом. Java можна легко розширити, оскільки вона заснована на моделі Object;

– незалежність від платформи - На відміну від багатьох інших мов програмування, включаючи C і C ++, коли Java компілюється, вона не компілюється під конкретну платформу, а в незалежний байт-код. Цей байт-код поширюється через веб та інтерпретується віртуальною машиною (JVM) на будь-якій платформі, на якій він працює;

– простота - Java розроблена так, щоб її було легко вивчити. Досить зрозуміти основну концепцію ООП Java;

– безпека - Завдяки захищеності Java, вона дозволяє розробляти системи, що не бояться вірусів;

– портативність – байт-код робить Java портативною. Код Java можна запускати на будь-яких платформах;

– надійність - Java має механізми обробки помилок часу виконання та помилок компіляції, що робить програми розроблені на Java ще більш надійними.

Існує досить багато фреймворків для розробки веб-застосунків на Java, та найпопулярнішим є Spring. Spring Framework - це зрілий, потужний та дуже гнучкий фреймворк, орієнтований на створення веб-додатків на Java[9]. Однією з основних переваг Spring є те, що він піклується про більшість аспектів низького рівня створення програми, щоб розробники могли насправді зосередитись на особливостях та логіці бізнесу.

Ще одним важливим моментом є те, що, хоча фреймворк є досить зрілим та добре налагодженим, він дуже активно підтримується та має процвітаючу спільноту розробників. Це робить його досить сучасним та привабливим для роботи з екосистемою Java.

Spring Framework має багато модулів, що дозволяє розробляти різноманітні веб-додатки. Для розробки додатків на основі мікросервісної архітектури Spring Framework надає бібліотеки Spring Cloud, що включають в себе велику кількість інших бібліотек для роботи з мікросервісами. Також Spring Cloud добре взаємодіє з бібліотеками Netflix OOS[10].

Для створення різноманітних Spring застосунків, компанія Spring надає зручний сервіс Spring Initializr (рисунок 2.11).

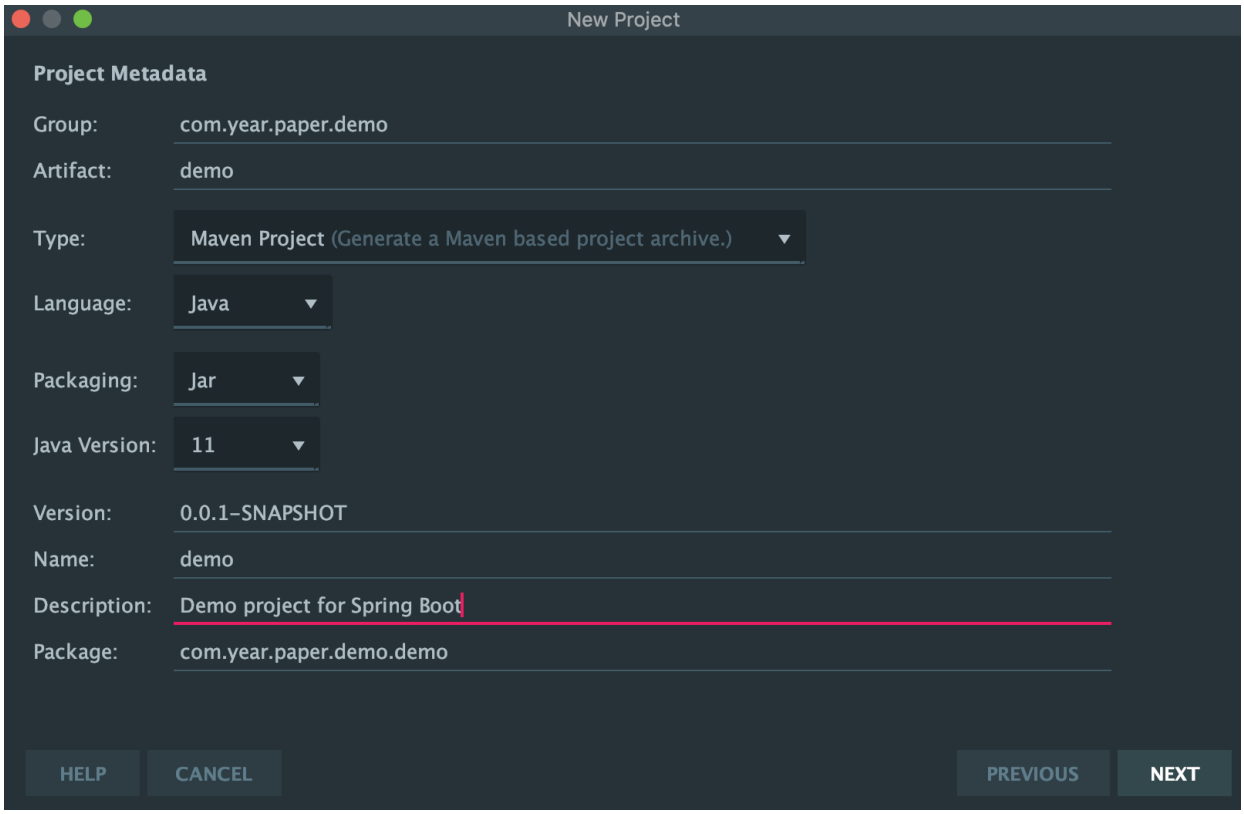


Рисунок 2.11 – Приклад створення нового Spring проекту

Після чого можливо підключити усі необхідні компоненти до проекту.

Як можна побачити на даному етапі уже існує можливість додати інтеграцію з Netflix OSS. У прикладі ми додаємо до проекту підтримку бібліотеки Netflix Eureka, Netflix Zuul, Netflix Ribbon[10] (рисунок 2.12).

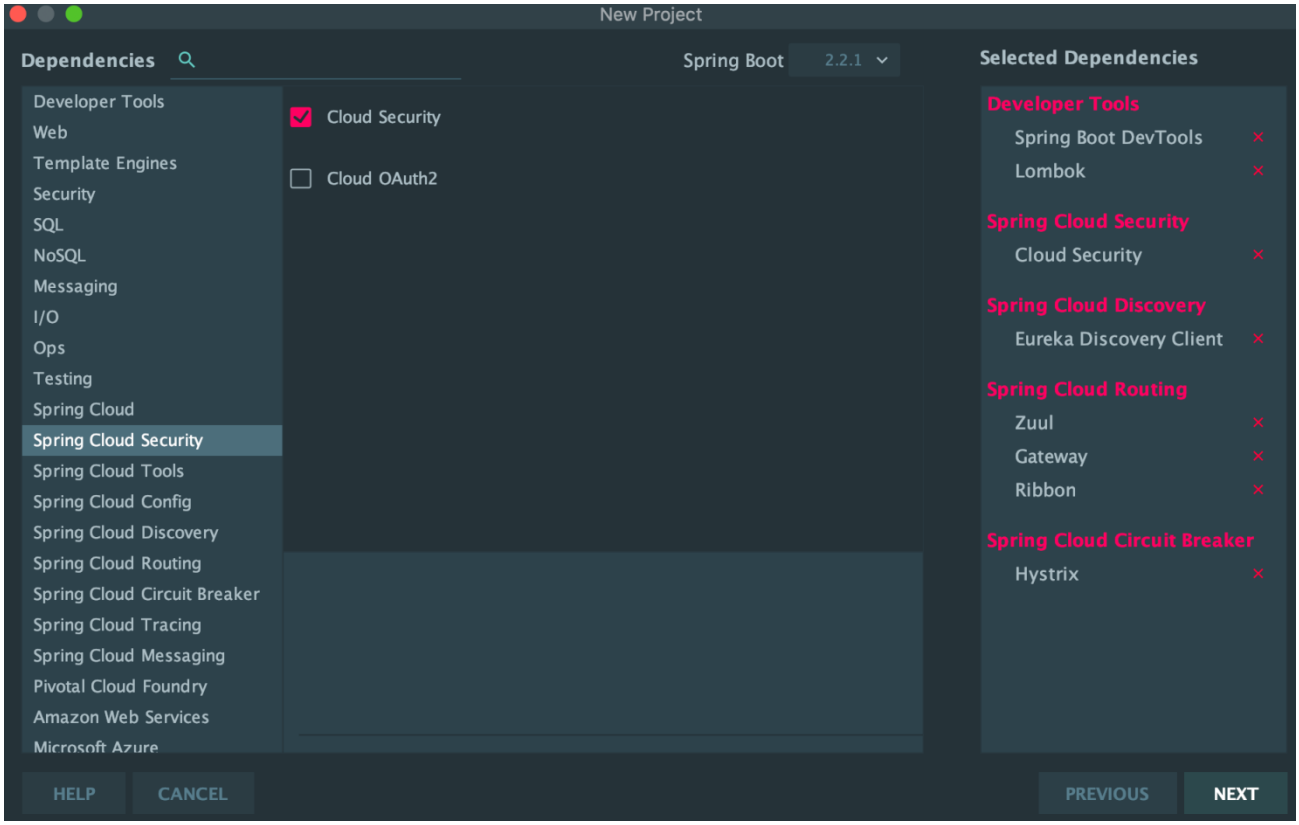


Рисунок 2.12 – Додавання залежностей до проекту за допомогою Spring Initializr

2.4.3 Ruby/Ruby on Rails

Ruby - інтерпретована мова програмування високого рівня. Володіє незалежною від операційної системи реалізацією багатопоточності, суворої динамічною типізацією, «збирачем сміття» і багатьма іншими можливостями, підтримують багато різних парадигм програмування, перш за все класово-об'єктну. Ruby був задуманий в 1993 році (24-го лютого) японцем Юкіхіро Мацумото, який прагнув створити мову, яка поєднує всі якості інших мов, що сприяють полегшенню праці програміста[11].

"Ruby on Rails" (або просто "Rails") - це фреймворк для розробки веб-додатків, написаний на мові програмування Ruby. З часів свого дебюту в 2004 році, Ruby on Rails досить швидко став одним з найпотужніших і популярних інструментів для створення динамічних веб-додатків. Rails використовується безліччю абсолютно різних компаній:

Airbnb, Basecamp, Disney, GitHub, Hulu, Kickstarter, Shopify, Twitter і Yellow Pages[11].

Що ж робить Rails таким чудовим? По-перше, Ruby on Rails - це на 100% відкритий вихідний код, доступний під MIT ліцензією, і, як наслідок, його завантаження та використання абсолютно безкоштовні. В результаті безліч завдань веб-програмування - таких як генерація HTML, створення моделей даних і маршрутизація URL - легкі в Rails, а результуючий код додатків короткий і легко читаємий.

Rails також дуже швидко адаптується до нових тенденцій в веб-технологіях. Наприклад, Rails був одним з перших, хто повністю реалізував архітектурний REST-стиль для структурування веб-додатків. І коли інші фреймворки успішно розробляють нові техніки, творець Rails, Девід Хайнмаєр, і робоча група Rails не соромляться використовувати їх ідеї[11]. Мабуть, найбільш яскравим і драматичним прикладом є злиття Rails і Merb (конкуруючий веб-фреймворк), завдяки якому Rails отримав модульний дизайн Merb, стабільний API і поліпшену продуктивність.

2.4.4 Node/Express

Як асинхронне подієве JavaScript-оточення, Node.js спроектований для побудови масштабованих мережних додатків. В застосунках Node.js для кожного з'єднання викликається функція зворотного виклику, однак, коли з'єднань немає Node.js засинає[12].

Node.js створений під впливом таких систем як Event Machine в Ruby або Twisted в Python. Але при цьому подієва модель, в ньому, використовується значно ширше, приймаючи event loop за основу оточення, а не в якості окремої бібліотеки. В інших системах ж завжди відбуваються блокування виклику, щоб запустити цикл подій.

Express - найпопулярніший веб-фреймворк Node.js що є базовою бібліотекою для ряду інших популярних веб-фреймворків Node,js. Цей фреймворк забезпечує механізми для таких речей як[13]:

- Написання обробників для HTTP/HTTPS запитів.
- Інтеграція з клієнтською частиною .
- Додання додаткових прошарків для додаткової обробки запитів.

Хоча сам Express досить мінімалістичний, розробники створили досить багато пакетів програмного забезпечення для вирішення практично будь-якої проблеми веб-розробки. Є бібліотеки для роботи з файлами cookie, сеансами, авторизацією користувачів, параметрами URL-адреси, даними POST, заголовками безпеки та багато іншого.

2.5 Висновок до розділу

В даному розділі було розглянуто основні шаблони для роботи з мікросервісною архітектурою, їх переваги та недоліки. Було проведено порівняння існуючих технологій для розробки застосунків з використанням мікросервісної архітектури. Огляд існуючих шаблонів та технологій дав змогу обрати найоптимальніший набір технологій для розробки веб-застосунку на основі мікросервісної архітектури.

Після вибору стеку технологій було розроблено загальну схему прецедентів, що дозволило описати варіанти використання системи та наочно зобразити функціонал системи. Детальний огляд реалізації застосунку на основі мікросервісних шаблонів з використанням обраних технологій буде проведено в наступному розділі.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Після проведення аналізу ключових особливостей мікросервісної архітектури на огляду наявних технологій у роботі з ними ми можемо перейти до процесу розробки системи. Система буде складатись з декількох ключових проектів, а саме:

– проект API Gateway, що буде виступати як API Gateway, а також як проксі сервер та буде містити клієнтську частину;

– сервер авторизації, що буде генерувати JWT токен по запиту та надавати засоби авторизації та автенфікації;

– Registry Server – сервіс, що буде виступати і як сервер конфігурації, і як Discovery сервер, і як система моніторингу системи;

– Events Server- звичайний мікросервіс, що буде надавати API для роботи з подіями.

3.1 API Gateway сервіс

Розробка API Gateway сервісу почнеться зі створення проекту Spring. Процес створення проекту Spring був описаний у другому розділі, тому тут ми опишемо лише залежності які необхідно підключити до даного проекту. Оскільки для збірки проекту ми використовуємо технологію Maven, усі залежності будуть описані як xml артефакти[17].

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
```

</dependency>

Опишемо необхідні нам залежності[14]:

- Spring-cloud-starter - дана бібліотека надає усі базові засоби для роботи з Spring Cloud, що представляє собою екосистему по роботі з мікросервісною архітектурою;
- Spring-cloud-starter-netflix-eureka-client – бібліотека Netflix OSS[16], що дозволяє оголосити наш додаток як Eureka Client, що дозволить у подальшому сервісу Registry знаходити даний сервіс та реалізовувати балансування навантаження на нього, тощо;
- Spring-cloud-security – бібліотека для підтримки авторизації та автентифікації. Доступ до сервісів системи буде надаватись через API Gateway, а отже JWT токен для авторизації теж буде проходити через API Gateway, щоб обробляти його правильним чином нам і потрібна дана бібліотека[18];
- Spring-cloud-starter-config – проекти побудовані на основі мікросервісної архітектури містять величезну кількість конфігураційних файлів. Для того щоб не ускладнювати і так складні сервіси і зберігати конфігураційні файли кожного з сервісів разом з самим сервісами, було запропоновано розробити окремий сервіс, що буде зберігати конфігураційні файли усіх сервісів. Це буде зручним рішенням, адже даний окремий сервіс буде надавати динамічно змінювати конфігурацію сервісів і не буде плутанини в пошуку необхідних конфігураційних файлів;
- Spring-cloud-starter-netflix-hystrix – бібліотека, що додає підтримку Netflix Hystrix[16], що відповідає за обробку затримок та відмов частин системи. Також дана бібліотека призначена для ізоляції точок доступу до сторонніх систем, правильну обробку каскадної відмови роботи сервісів та забезпечення стійкості системи в цілому;

– Spring-cloud-starter-netflix-zuul – бібліотека, що в цілому реалізує шаблон API Gateway, та у тісному зв’язку з іншими бібліотеками відповідає за безпеку, моніторинг сервісів, динамічне зв’язування сервісів, стійкість системи, балансування навантаження та багато чого іншого[16];

– Spring-cloud-starter-netflix-ribbon – бібліотека, що відповідає за міжпроцесорну комунікацію(віддалені виклики сервісів) та реалізовує шаблон балансування навантаження[16].

Точкою входу в проект Spring є клас, що позначений анотацією @SpringBootApplication. В нашому випадку клас також буде позначений також анотаціями @EnableZuulProxy, @EnableDiscoveryClient, @EnableConfigurationProperties.

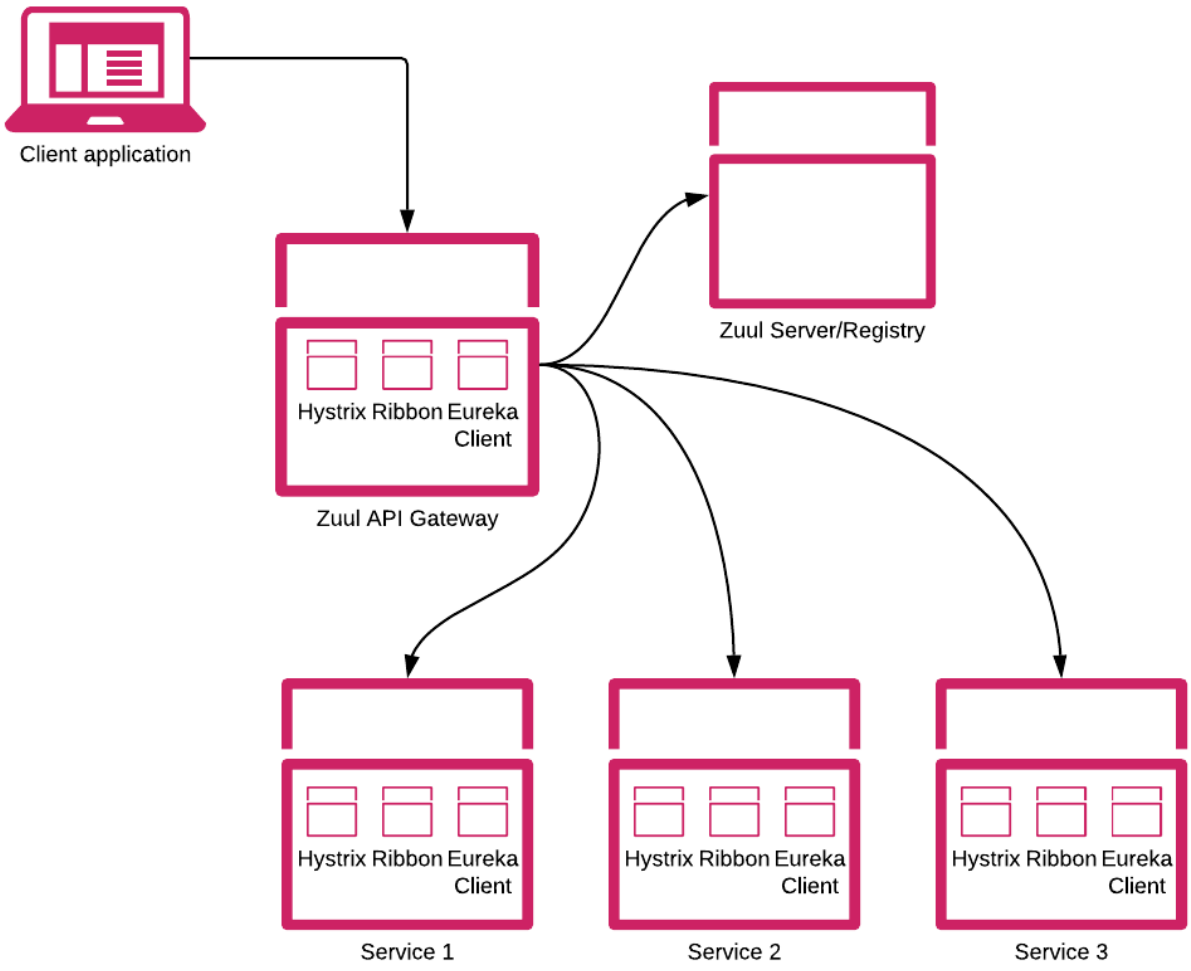


Рисунок 3.1 – Діаграма роботи API Gateway

Анотація @EnableZuulProxy позначає даний проект як проксі Zuul. Це дозволить нам робити декілька речей. По-перше це переадресувати усі запити зроблені до даного API Gateway на відповідні мікросервіси. Тобто клієнтській частині не потрібно знати адреси кожного мікросервісу щоб зробити запит до нього. Враховуючи те що мікросервісів може бути величезна кількість і усі вони можуть бути на різноманітних портах, а ще може бути запущено декілька екземплярів одного сервісу і потрібно якимось чином балансувати навантаження на нього та й робити запити по різним портам для доступу до одного й того ж сервісу не надто хочеться, це є дуже зручним. По-друге Zuul дозволяє використовувати фільтрі для роботи з запитами, існують так звані pre-filters та post-filters, що дозволяють обробляти запит до його відправки та після його отримання. Zuul добре працює разом з Hystrix та Ribbon.

Загальна схема така: запити зроблені до Zuul сервісу переадресуються до відповідних мікросервісів. Zuul використовує Ribbon для того щоб дізнатись до якого саме сервісу потрібно переадресовувати запит і яка його фізична адреса, а також для балансування навантаження. При цьому усі запити проходять через Hystrix, що дозволяє нам збирати статистику та мати метрику помилок і відповідно обробляти самі помилки. Діаграму роботи шаблону API Gateway зображено на рисунку 3.1.

```
zuul:
  routes:
    echo:
      path: /myusers/**
      serviceId: myusers-service
      stripPrefix: true

hystrix:
  command:
    myusers-service:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: ...

myusers-service:
  ribbon:
    NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList
    listOfServers: https://example1.com,http://example2.com
    ConnectTimeout: 1000
    ReadTimeout: 3000
    MaxTotalHttpConnections: 500
    MaxConnectionsPerHost: 100
```

Рисунок 3.2 – Приклад конфігураційного файлу

Для базового налаштування Zuul, Hystrix та Ribbon, нам необхідно модифікувати базовий конфігураційний файл проекту application.yml. Приклад такого файлу зображений на рисунку 3.2.

Тут ми бачимо приклад базової конфігурації Zuul, Ribbon та Hystrix. Значення serviceId, це id сервісу, що був зареєстрований за допомогою Eureka сервера, а нашому випадку Registry, тому відпадає необхідність записувати адрес сервісу повністю. А значення path, це шлях по якому буде доступне API сервісу з id serviceId. На рисунку також зображені налаштування для Hystrix та Ribbon, але поки не будемо на них зупинятись.

В цілому це є дуже зручним адже на клієнтській частині не потрібно думати про переадресації запитів, обробляти CORS та автентифікацію. Все це лягає на Zuul.

Анотація @EnableDiscoveryClient реєструє сервіс для Netflix Eureka. Це означає, що даний сервіс буде надавати метаінформацію про себе, таку

як хост, порт, url для перевірки життєдіяльності сервісу, url домашньої сторінки, тощо до серверу Eureka, що у нашому випадку буде проектом Registry.

Для роботи Eureka також потрібно додати певні налаштування до конфігураційних файлів.

```
eureka:
  client:
    enabled: true
    healthcheck:
      enabled: true
    fetch-registry: true
    register-with-eureka: true
    instance-info-replication-interval-seconds: 10
    registry-fetch-interval-seconds: 10
  instance:
    appname: events_slavatory_gateway
    instanceId: events_slavatory_gateway:${spring.application.instance-id:${random.value}}
    lease-renewal-interval-in-seconds: 5
    lease-expiration-duration-in-seconds: 10
    status-page-url-path: ${management.endpoints.web.base-path}/info
    health-check-url-path: ${management.endpoints.web.base-path}/health
```

Рисунок 3.3 – Конфігураційний файл application.yml

```
eureka:
  instance:
    prefer-ip-address: true
  client:
    service-url:
      defaultZone: http://admin:admin@localhost:8761/eureka/
```

Рисунок 3.3 – Конфігураційний файл application-dev.yml

Як можемо побачити, в конфігураційному файлі ми вказуємо шлях до серверу Eureka/Discovery Server. Завдяки цьому даний сервіс буде знати куди саме йому необхідно відправляти свої мета дані.

Далі коротко опишем структуру API Gateway. Проект буде мати наступну структуру папок рисунок 3.4.

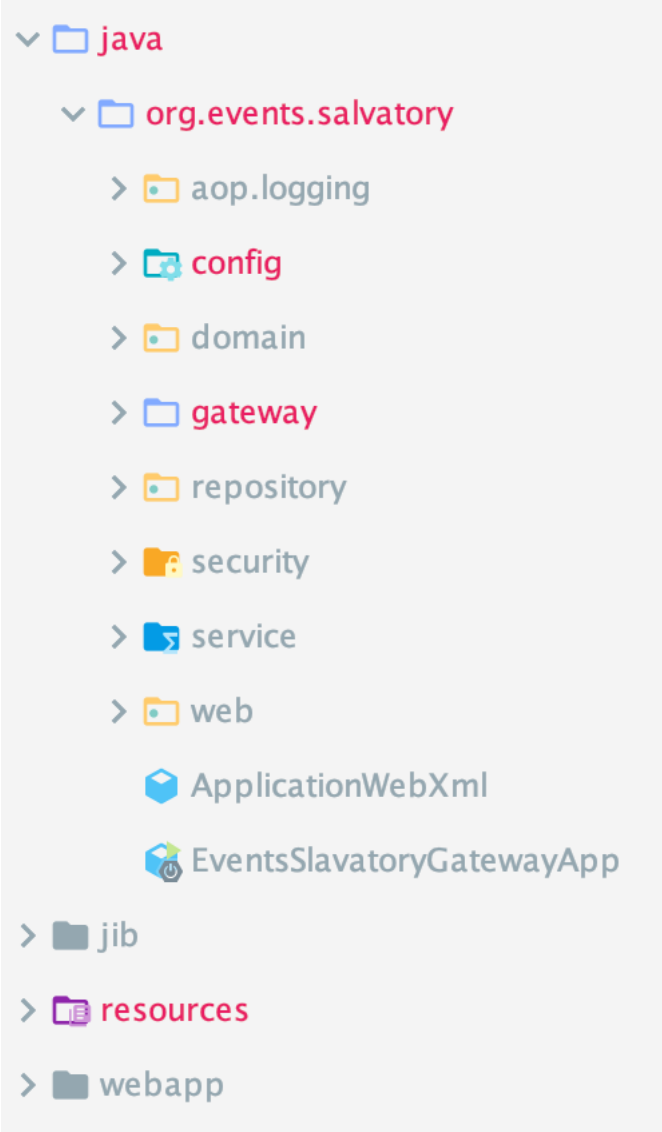


Рисунок 3.4 – Структура папок проекту API Gateway

– Aop.logging – пакет, що містить класи, що відповідають за логування;

– Config – пакет, що містить конфігураційні класи, один з найважливіших пакетів, так як містить усі конфігурації сервісу;

– Domain – пакет з допоміжними класами;

– Gateway – містить Zuul фільтри, що контролюють доступ до мікросервісів;

– Security – містить класи, що відповідають за автентифікацію та авторизацію користувача та роботу з UAA сервером;

– Web – містить контролери та фільтри для оновлення JWT токену у разі його застаріння;

– Resource – пакет, що зберігає конфігураційні файли;

– Webapp – пакет, що містить клієнтську частину коду.

Розглянемо більш детально кожен з ключових конфігураційних класів пакету config. Структура класів пакету config зображено на рисунку 3.5.

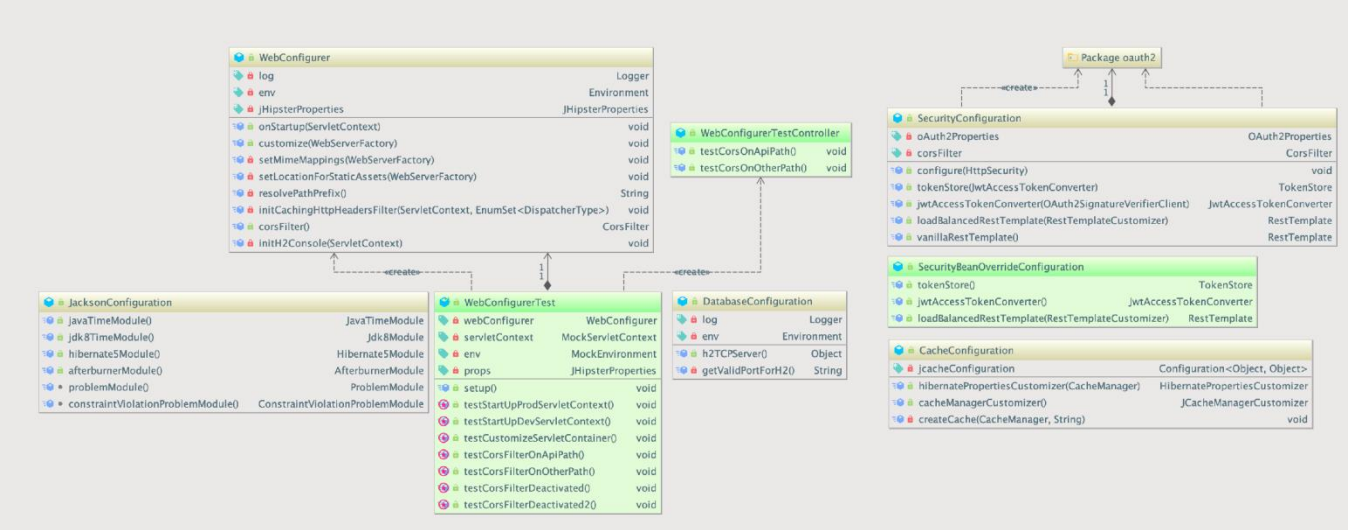


Рисунок 3.5 – Діаграма класів пакету config

– WebConfigurer – клас, що задає базові веб-налаштування, такі як кешування хедерів запиту, конфігурація CORS фільтру тощо;

– DatabaseConfiguration - клас, що містить конфігурацію баз даних, та відповідає за ініціалізацію H2 бази даних при розробці;

– SecurityConfiguration – клас, що відповідає за налаштування безпеки. В даному класі ми конфігуруємо які шляхи доступні для звичайного користувача, а які для адміністратора. Задаємо алгоритм конвертації JWT токену, конфігуруємо клас, що використовується для того щоб робити запити між мікросервісами і т.д.

Решта конфігураційних класів є другорядними. Далі розглянемо класи пакету web що зображені на рисунку 3.6.

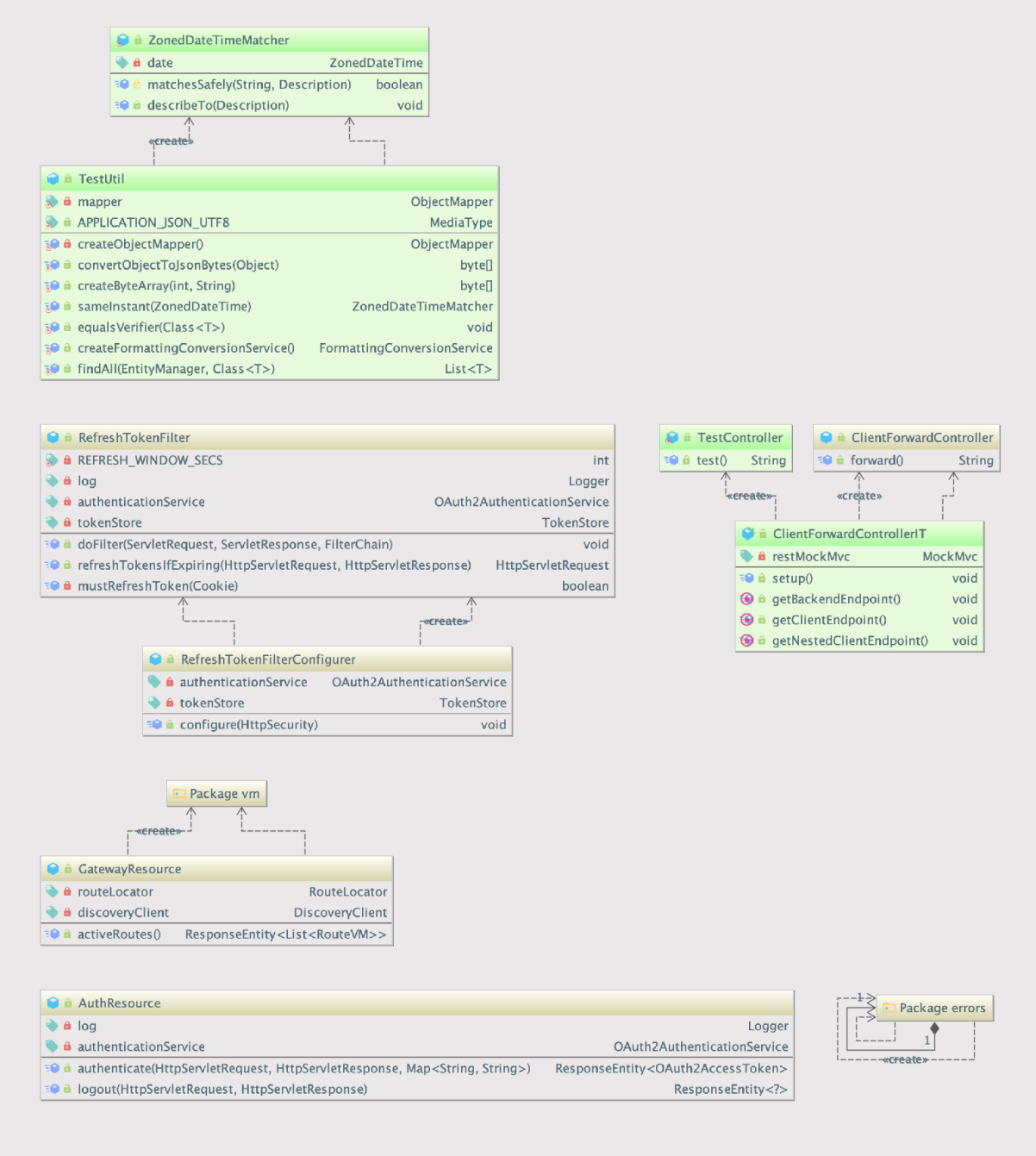


Рисунок 3.6 – Діаграма класів пакету web

Тут варто виділити класи контролери GatewayResource, ClientForward, AuthResource, що відповідають за обробку запитів клієнта і віддачу певної відповіді на запит.

Пакет gateway містить Zuul фільтри, що обробляють запити при отриманні і перед відправкою (рисунок 3.7).

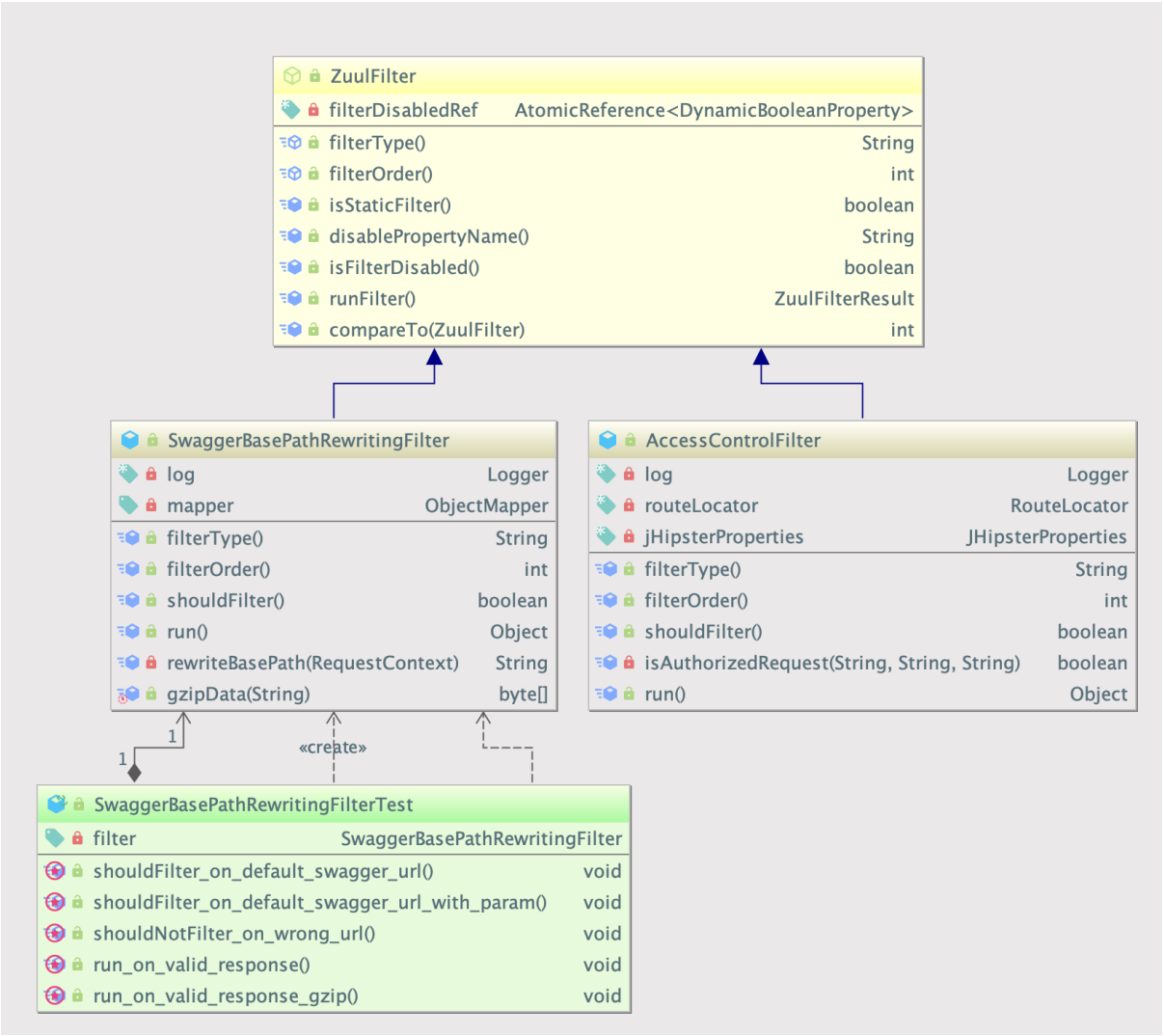


Рисунок 3.7 – Діаграма класів пакету gateway

Наступним ключовим пакетом для сервісу є пакет security (рисунок 3.8). Даний пакет містить класи з логікою роботи з JWT токеном та сервером авторизації в цілому.

Клас UaaTokenEndpointClient реалізує функції клієнту доступу до UAA серверу для отримання та валідації JWT токену та даних про користувача.

Клас OAuth2Cookies зберігає значення JWT токену та додає їх до куки у разі успішної автентифікації.

OAuth2AuthenticationService клас керує процесом автентифікації, може робити запити на отримання JWT токену, запит на оновлення

токену та чистити усі куки, що зберігають інформацію, що відноситься до користувача, тим самим вилогінюючи користувача.

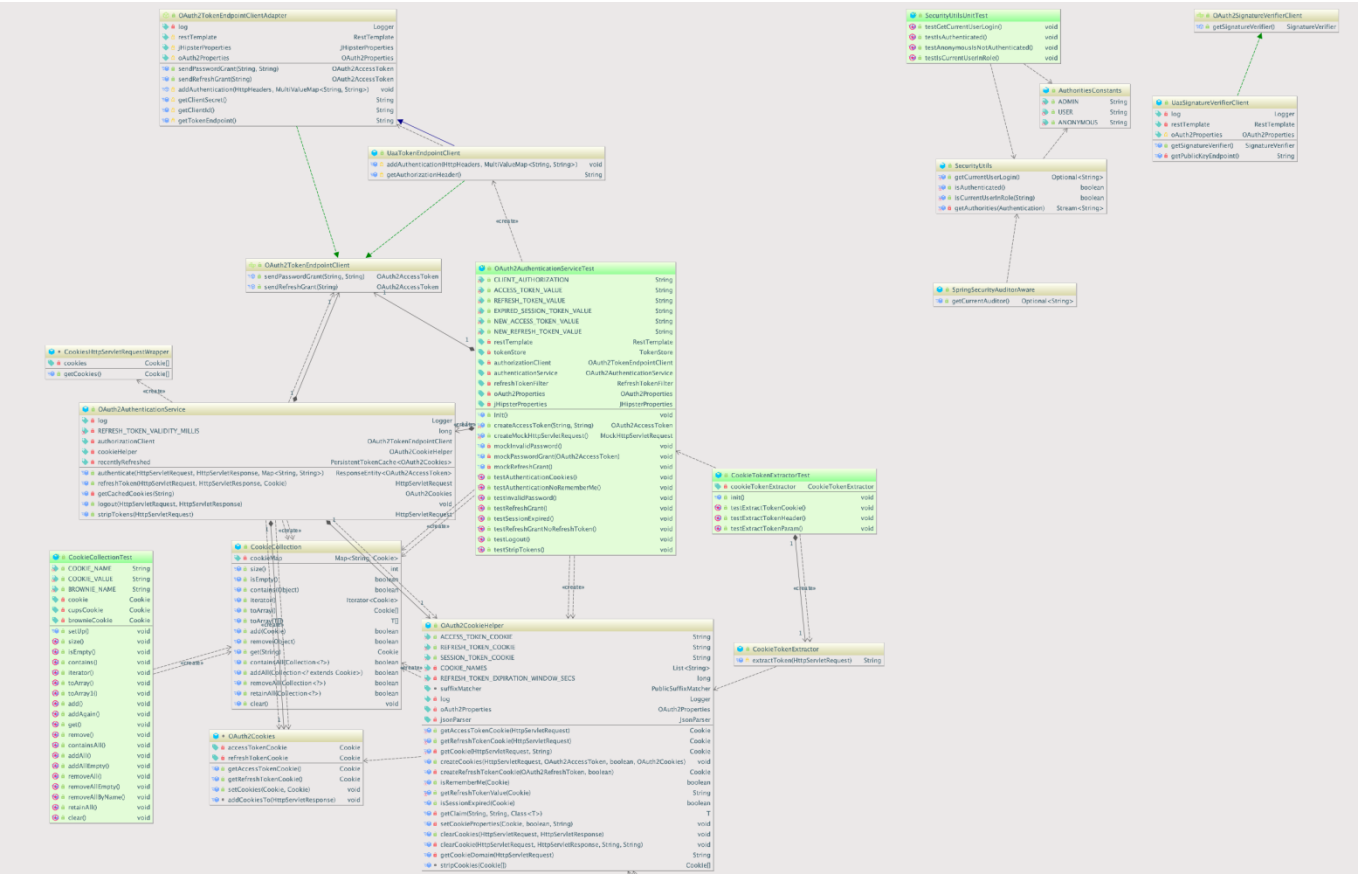


Рисунок 3.8 – Діаграма класів пакету security

3.2 UAA сервер

Наступним сервісом для реалізації буде UAA сервіс. UAA(User Account and Authentication) сервіс буде відповідати за процеси авторизації та автентифікації та захищати систему використовуючи OAuth2 протокол.

OAuth2 це протокол який дозволяє системі, що його реалізує, відповідати таким запитам як:

– мати центральний механізм автентифікації. Оскільки мікросервіси це незалежні і автономні програми, ми хочемо бути впевнені, що запит користувача не буде оброблятись різними застосунками з, можливо, різними системами захисту;

– відсутність стану. Тобто дані про те чи автентифікований користувач не зберігаються в сесії. Для цього використовується JWT токен, що може зберігатись в куках або в локальному сховищі[20]. І оскільки основна з головних переваг мікросервісної архітектури це легка масштабованість, обране рішення в підтримці безпеки не має впливати на цю ключову перевагу;

– відмінність у механізмах отримання доступу до системи для користувачів та машин. Використання мікросервісної архітектури призводить до створення великого багатоцільового центру обробки даних з різних доменів та ресурсів, тому виникає необхідність обмежувати доступ різних клієнтів, таких як нативні програми, різні SPA і т.д;

– роздільність контролю доступу. При збереженні централізованих ролей існує потреба в налаштуванні різних політик контролю доступу для кожного мікросервісу. Мікросервіс не повинен усвідомлювати відповідальність за розпізнавання користувачів і повинен просто авторизувати вхідні запити;

– бути добре захищеною від атак;

– бути легко масштабованою.

В цілому алгоритм роботи з сервером автентифікації можна описати наступним чином:

– кожен запит до будь-якого ресурсу по URL виконується за допомогою клієнту;

– клієнт це деяка абстракція, що може бути і як http клієнтом, і як REST клієнтом і власне будь-що що може виконувати запити;

– клієнт також може використовуватись для автентифікації користувача, для прикладу за допомогою ajax запиту з клієнтської частини;

– кожен мікросервіс, включаючи UAA сервер, що надає певні ресурси за певними URL являється так званим ресурсним сервером;

– стрілки синього кольору зображують запити автентифікації до серверу автентифікації UAA;

– стрілки зеленого кольору зображують запити зроблених з клієнтів до ресурсних серверів;

– UAA сервер є комбінацією серверу авторизації та ресурсного серверу;

– UAA сервер є власником усіх даних в середині проекту та контролює доступ до них;

– клієнти, що отримують доступ до ресурсів з автентифікацією користувача, автентифікуються за допомогою "password grand" з ідентифікатором клієнта(cliend id) та секретом(secret), що безпечно зберігаються у файлах конфігурації Registry;

– клієнти, що отримують доступ до ресурсів без даних користувача, автентифікуються за допомогою "client credentials grant";

– кожен клієнт визначається в конфігураційних файлах UAA серверу.

Схематично схема роботи серверу авторизації та автентифікації зображена на рисунку 3.9.

В цілому UAA сервер виконує такі три функції:

– надає доступ до ресурсу, що містить дані про користувача та акаунти;

– реалізує інтерфейс AuthorizationServerConfigurerAdapter для реалізації OAuth2 та визначає базові клієнти, такі як web_app, internal, тощо;

– генерує JWT публічний ключ, що має використовуватись усіма іншими мікросервісами.

Коли мікросервіс завантажується, зазвичай, він очікує, що сервер UAA вже готовий надати свій відкритий ключ. Служба спочатку викликає /oauth/token_key, щоб отримати відкритий ключ та налаштувати його для підписання токену (JwtAccessTokenConverter)[19].

Для запитів користувача на вхід надсилається запит до API Gateway /auth/login. Ця кінцева точка використовує OAuth2TokenEndpointClientAdapter для надсилання запиту до UAA, що підтверджує автентифікацію з наданням пароля. Оскільки цей запит відбувається та обробляється в API Gateway, client id та secret не зберігаються в коді клієнта і є недоступними для користувачів. API Gateway повертає новий файл cookie, що містить JWT token, і цей файл cookie надсилається з кожним запитом, виконаним від клієнта, до серверної частини проекту. Там він валідується і у разі успішної валідації дозволяє використовувати певні ресурси та обробляти їх[19].

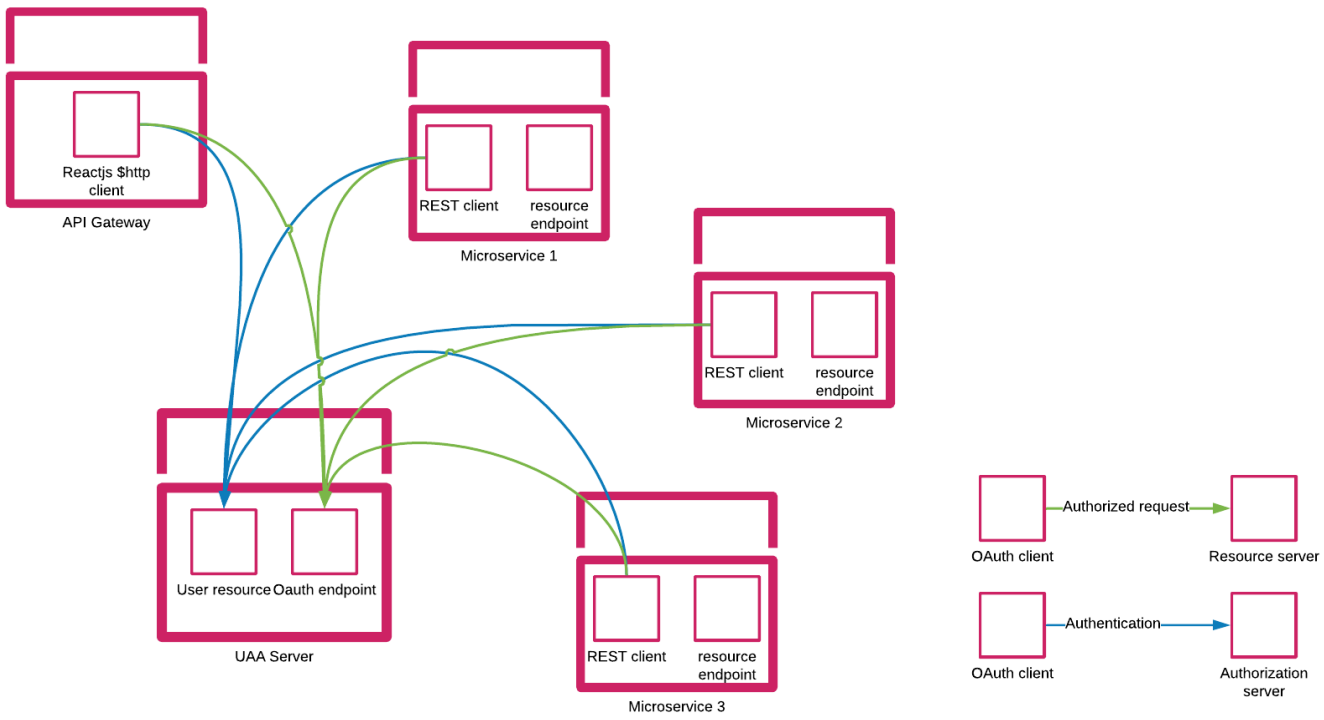


Рисунок 3.9 – Схема роботи UAA серверу

Тож розглянемо розроблюваний додаток поближче. Залежності, що потрібно додати до проекту, схожі на ті, що й для API Gateway. Хіба, що тут необхідно також додати такі додаткові залежності:

– spring-boot-starter-data-jpa.
– spring-boot-starter-mail..
– spring-boot-starter-security
– spring-security-jwt.
– spring-security-data.

Вони допоможуть нам у роботі з базами даних та з JWT токеном. Даний проект власне має аналогічну структуру папок до API Gateway.

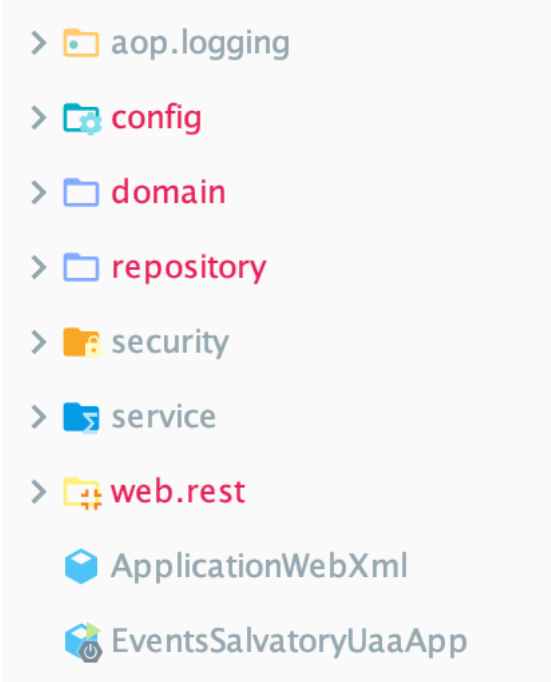


Рисунок 3.10 – Структура проекту UAA серверу

Але у даному випадку ми маємо такі ключові пакети.

Config – в даному випадку, пакет містить конфігураційні файли для серверу авторизації.

Domain – містить класи сутності, для роботи з базою даних. Насправді це не що інше як POJO класи, що представляють собою дані, що можна зберігати до бази даних. Сутність представляє собою таблицю, що буде зберігатись у базі даних. В той час кожен екземпляр сутності представляє собою рядок у таблиці. Схематично пакет зображений на рисунку 3.11.

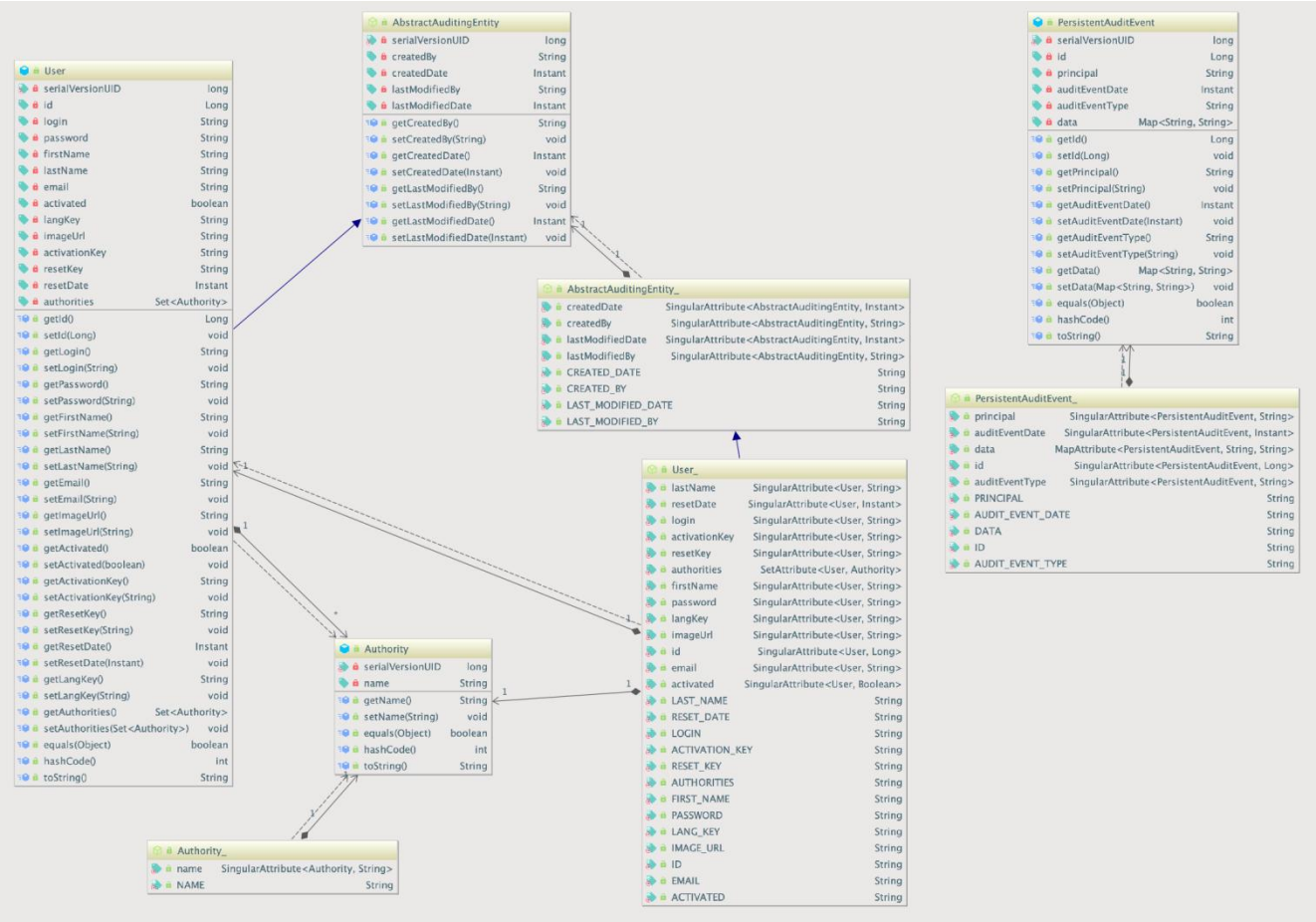


Рисунок 3.11 – Діаграма класів пакету domain

Repository – пакет, що містить класи Spring Data JPA. Цей модуль спрощує побудову додатків, що працюють з базами даних, та дозволяють за допомогою класів та конфігураційних файлів описувати сутності баз даних та роботу з ними. Діаграму класів пакету repository для проекту UAA server зображено на рисунку 3.12.

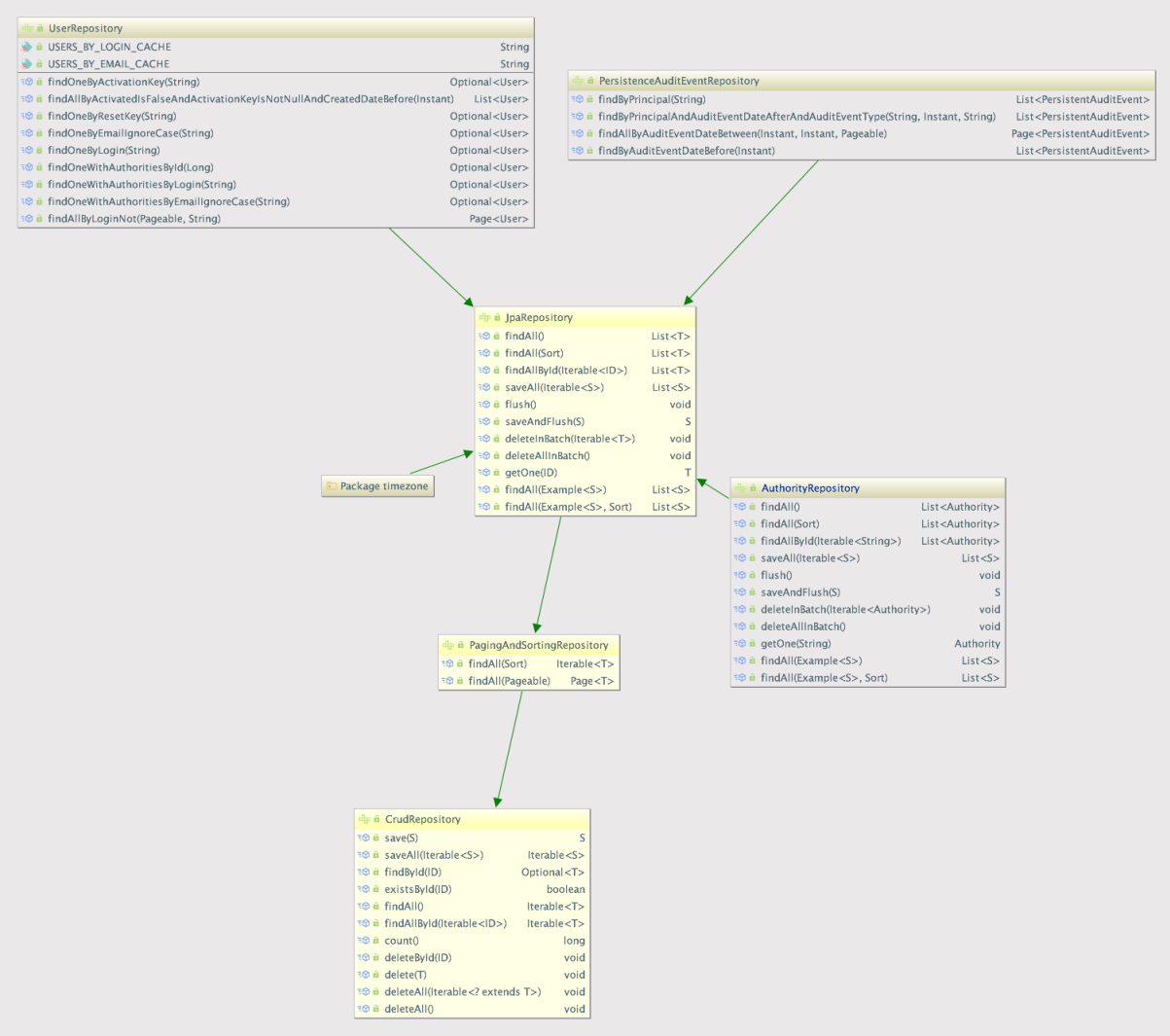


Рисунок 3.12 – Діаграма класів пакету repository

Security – містить допоміжні класи, які використовуються для конфігурації в пакеті config.

Service – містить сервіси для роботи з користувачем. Надає таку функціональність як реєстрація, авторизація, отримання даних користувача, тощо. Діаграму класів пакету service для проекту UAA server зображено на рисунку 3.13.

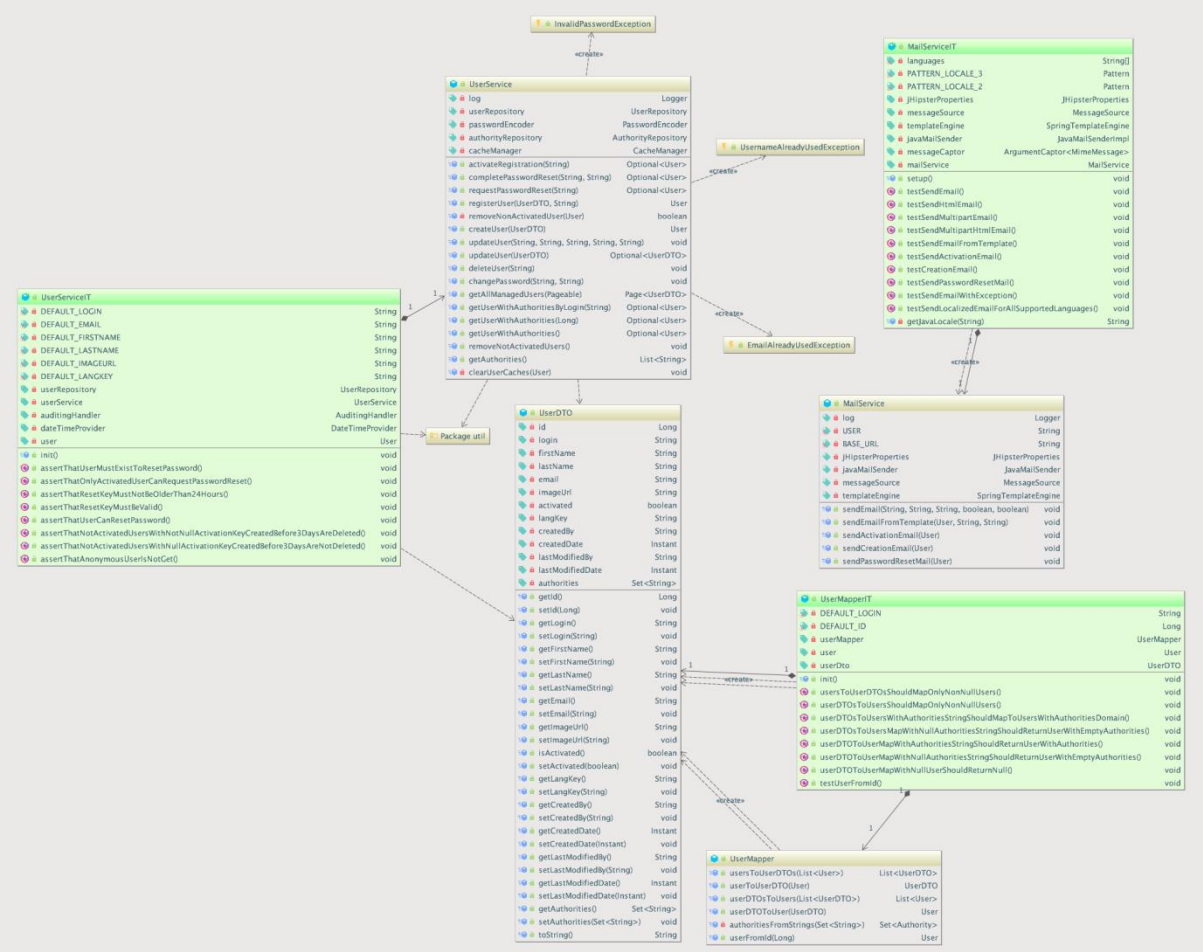


Рисунок 3.13 - Діаграма класів пакету service

3.3 Registry сервер

Registry сервер це мікросервіс, що виконує три основні функції:

– являється Eureka сервером, що виступає в ролі Discovery server. Саме завдяки Eureka застосунок вирішує питання зв’язування, балансування навантаження та масштабованості усіх мікросервісів;

– являться сервером конфігурації Spring Configuration Server, що надає конфігураційні файли для усіх мікросервісів під час виконання;

– являється сервером адміністрування, моніторингу та обробки сервісів.

Головний клас даного проекту позначений декількома анотаціями. Анотація EnableEurekaServer позначає проект як сервер Eureka і означає, що даний проект буде виступати як Discovery server. EnableConfigServer позначає як сервер конфігурацій, тобто усі мікросервіси зможуть зчитувати конфігураційні файли з єдиного сховища, а ще існує можливість динамічної зміни самих конфігураційних файлів.

3.4 Events сервіс

Events сервіс це невеликий мікросервіс, що якраз буде надавати системі API для роботи з подіями. Для створення даного мікросервісу була створена модель даних для сервісу.

Основними сутностями моделі є Організація, Подія та Локація. Розглянемо кожну з сутностей. ER-діаграма проектованої сутності зображена на рисунку 3.14. Розглянемо детальніше кожну з сутностей.

Сутність Організація має наступні поля.

- Id – первинний ключ таблиці.
- Name – поле, що містить ім’я організації.
- Description – поле, що містить опис організації.
- Rating – поле, що містить рейтинг організації.
- CreatorName – поле, що містить назву засновника організації.
- CreationDate – поле, що містить дату створення організації.

Дана сутність представляє собою організацію, що проводить певну подію.

Сутність Подія має наступні поля.

- Id – первинний ключ таблиці.
- Name – поле, що містить назву події.
- Description – поле, що містить опис події.
- Price – поле, що містить ціну події.

- Address – поле, що містить адресу події.
- Image – поле, що містить шлях до картинки.
- Rating – поле, що містить оцінку події.
- eventType – поле, що містить тип події.
- creationDate – поле, що містить дату створення події.
- startDate – поле, що містить дату початку події.
- endDate – поле, що містить дату закінчення події.

Дана сутність являється основною у даному мікросервісі.

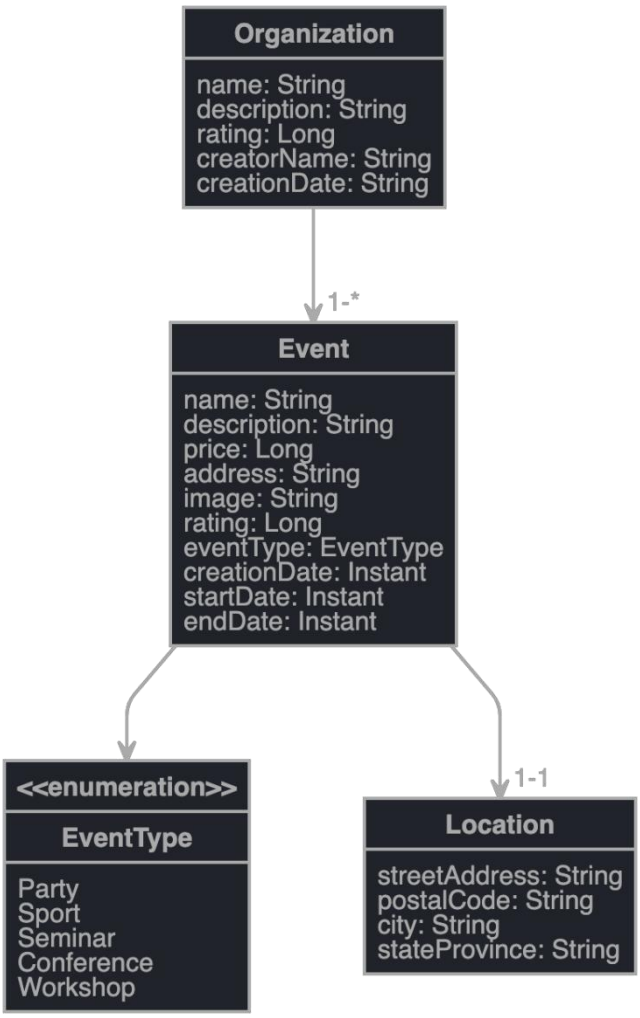


Рисунок – 3.14 – ER-діаграма сервісу

Останньою сутністю являється Локація, що зберігає у собі дані про розміщення події і містить наступні поля:

- Id – первинний ключ таблиці.
- StreetAddress – поле, що містить адресу вулиці.
- PostalCode – поле, що. Містить поштовий код.
- City – поле, що містить назву міста.
- StateProvince – поле, що містить назву району.

На основі розробленої моделі даних, ми маємо змогу розробити програмну реалізацію даної моделі використовуючи ORM(Object-relational mapping). ORM - це механізм, який дає змогу звертатися та керувати об’єктами, не враховуючи, як ці об’єкти зберігаються. ORM це ідея спроможності писати запити до бази даних будь-якої складності, використовуючи об'єктно-орієнтовану парадигму бажаної мови програмування. В даному випадку Spring надає зручний модуль для роботи з ORM – Spring Data. Даний модуль дозволяє за допомогою анотацій визначати сутності баз даних. Як можна побачити на рисунку нам необхідно створити так званий POJO клас. Після чого позначити його анотаціями @Entity та @Table, що позначать його як сутність та таблицю і допоможуть модулю Spring Data відповідним чином обробити даний клас і створити на його основі відповідну таблицю у будь-якій базі даних. Приклад такого класу зображений на рисунку 3.15.

```java
@Entity
@Table(name = "event")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Event implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "description")
    private String description;

    @Column(name = "price")
    private Long price;

    @Column(name = "address")
    private String address;

    @Column(name = "image")
    private String image;

    @Column(name = "rating")
    private Long rating;

    @Enumerated(EnumType.STRING)
    @Column(name = "event_type")
    private EventType eventType;

    @Column(name = "creation_date")
    private Instant creationDate;

    @Column(name = "start_date")
    private Instant startDate;

    @Column(name = "end_date")
    private Instant endDate;

    @OneToOne(fetch = FetchType.LAZY)

    @JoinColumn(unique = true)
    private Location location;

    @ManyToOne(fetch = FetchType.LAZY)
    @JsonIgnoreProperties("events")
    private Organization organization;
```

Рисунок 3.15 – Клас Event

Усі ORM сутності знаходяться у пакеті domain і його загальна структура має наступний вигляд (рисунок 3.16).

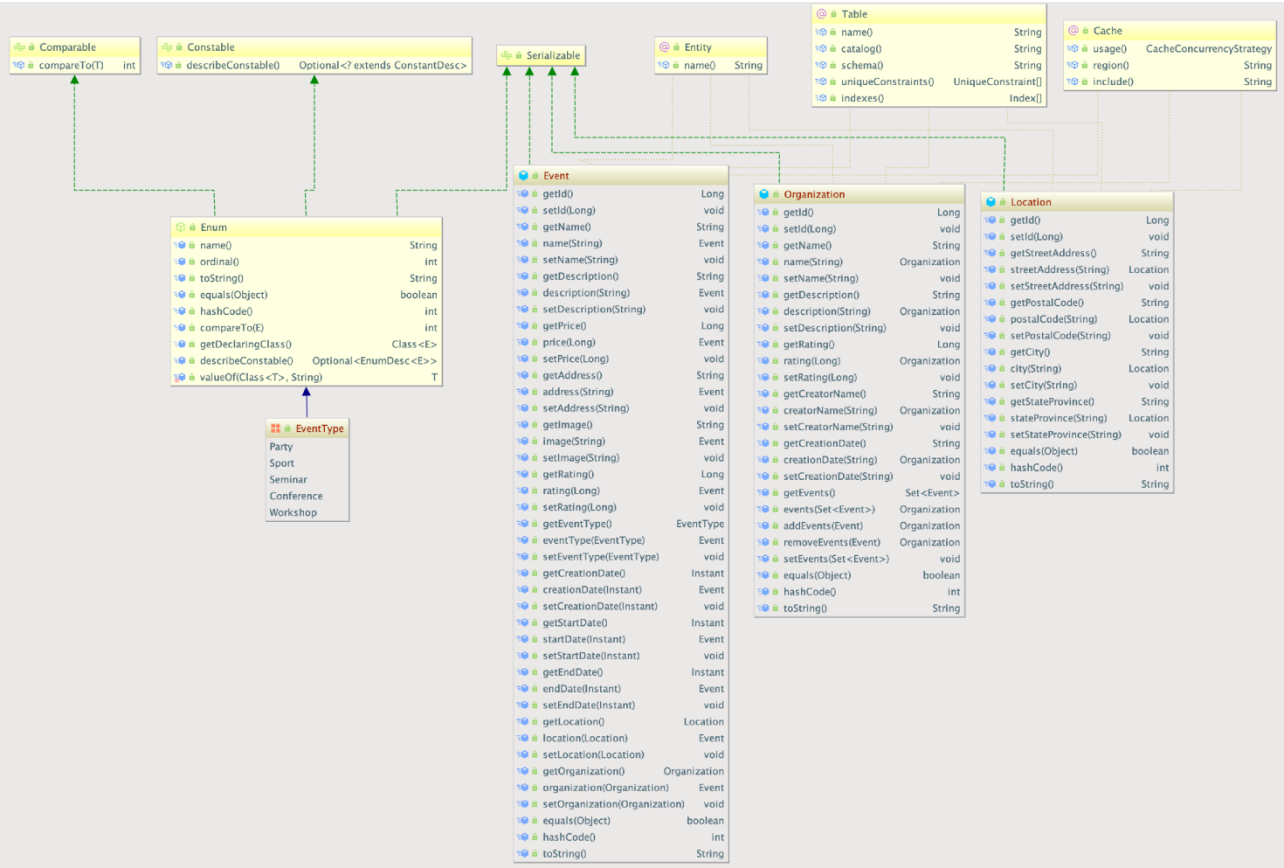


Рисунок 3.16 – Діаграма класів пакету domain

Для роботи роботи з даними ми маємо модуль доступу до даних під назвою repository. Для доступу до даних будемо використовувати Spring Data Repositories. Даний модуль дозволяє створювати запити до баз даних за допомогою програмного коду.

Для цього необхідно реалізувати інтерфейс JpaRepository. Після чого просто називаючи відповідним чином методи можливо створювати запити до бази даних. Скажемо аналогом до наступного запиту до бази даних select e from Event e where e.name = ?1 and e.address = ?2 буде наступний програмний код[15]. Приклад коду зображений на рисунку 3.17.

```
@Repository
public interface EventRepository extends JpaRepository<Event, Long> {

    List<Event> findByNameAndAddress(final String name, final String address);

}
```

Рисунок 3.17 – Приклад роботи з інтерфейсом JpaRepository

3.5 Загальний опис роботи системи

Для запуску системо необхідно розгорнути усі 4 мікросервіси. Першим варто розгортати Registry мікросервіс, адже він являється сервером конфігурації і всі інші мікросервіси будуть отримувати конфігураційні дані саме з нього. Registry сервер розгортається на порту 8761. Сервер API Gateway розгортається на порту 8080. UAA сервер розгортається на порту 9999, а мікросервіс Events service розгортається на порту 8081.

Розглянемо ближче роботу з розробленою системою. Коли користувач вперше завітає до вебсистеми, він побачить домашню сторінку зі списком подій(рисунок 3.18).

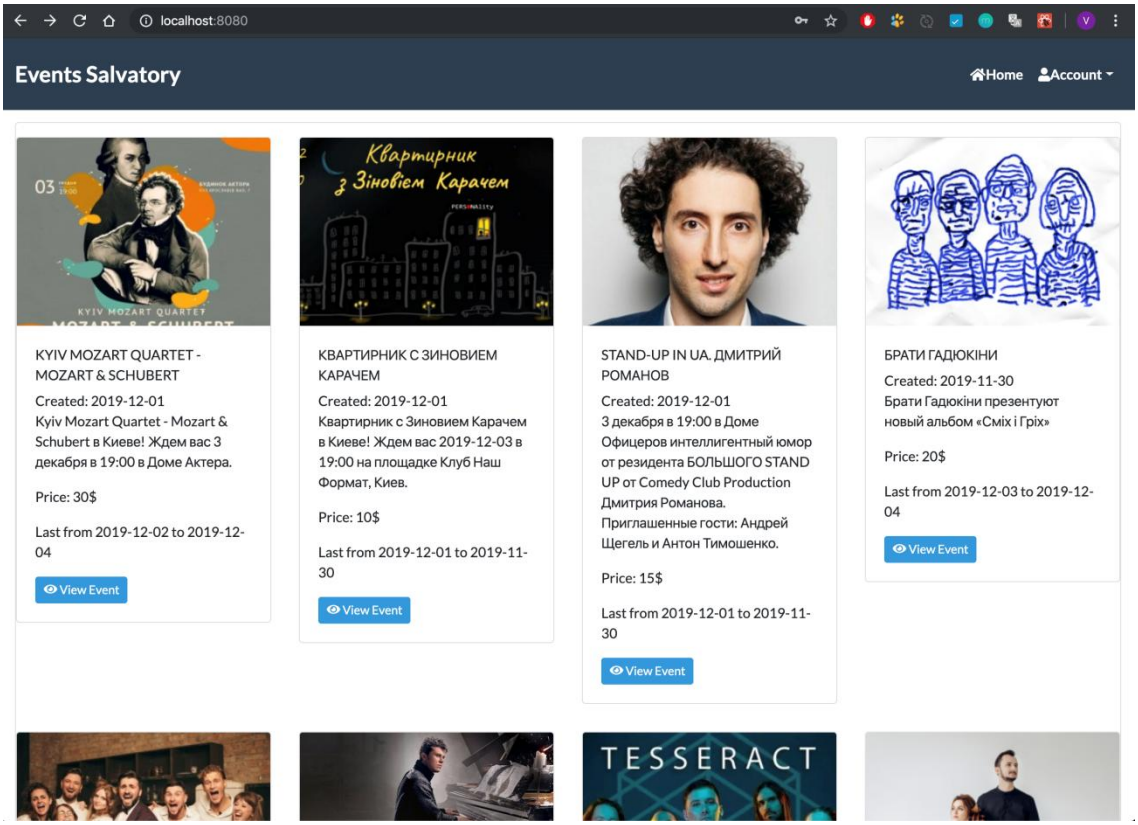


Рисунок 3.18 – Домашня сторінка застосунку

Далі користувач має змогу перейти зареєструвати нового користувача(рисунок 3.19).

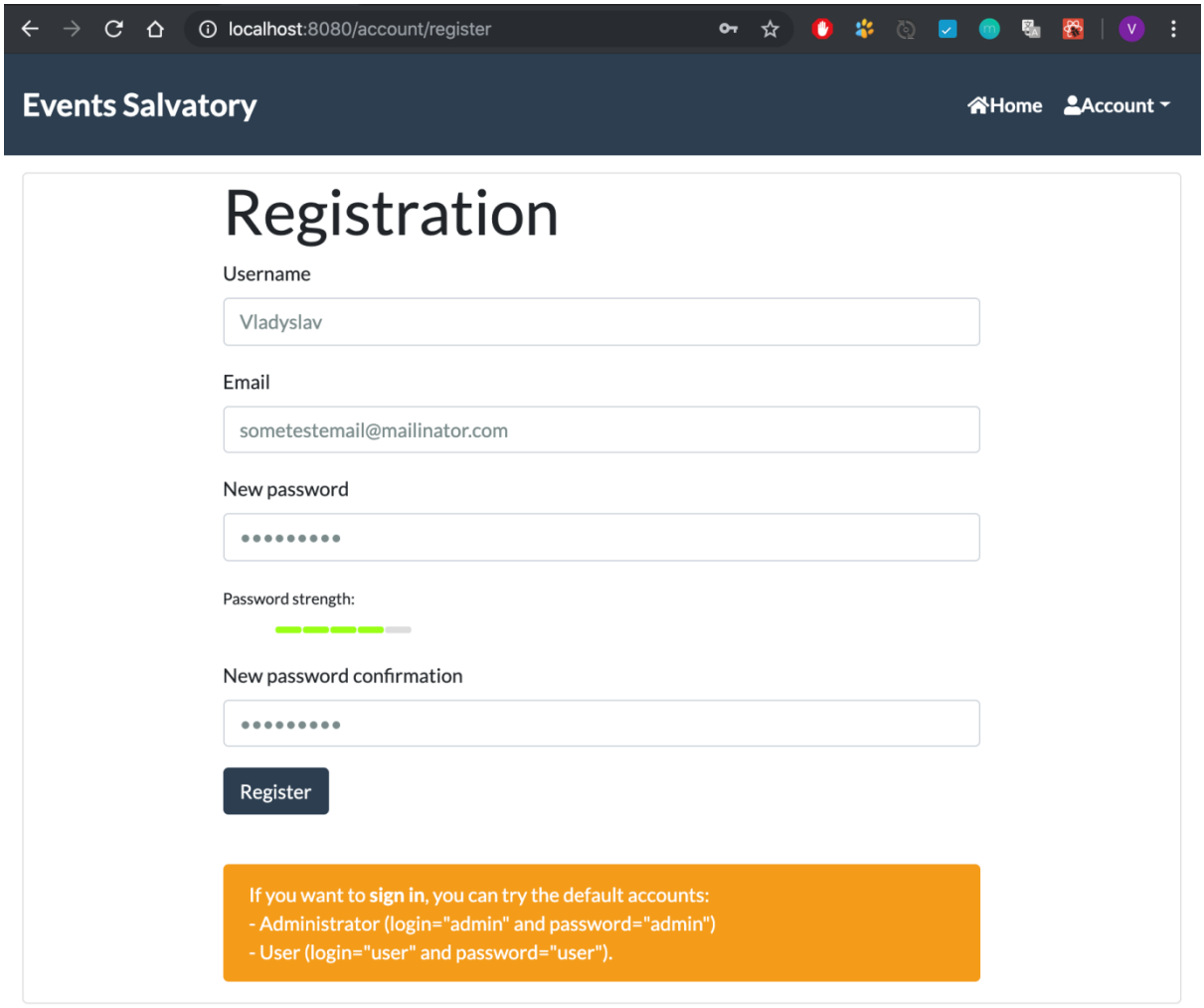


Рисунок 3.19 – Форма реєстрації

Заповнивши необхідні поля та натиснувши на кнопку Register буде зроблений POST запит. Як можна побачити з рисунка 3.20, запит буде зроблений за адресою http://localhost:8080/services/eventssalvatoryuaa/api/register. Власне тут ми і бачимо як працює API Gateway, насправді запит буде зроблений до UAA серверу, що знаходиться на порті 9999, але API Gateway за допомогою Registry сервера, дізнається, що існує деякий UAA сервер, що має назву eventsSalvatoryUaa, який надає API для реєстрації користувача за шляхом /api/register, додасть до цього всього загальний суфікс /services в результаті чого і вийде описана вище адреса. Відповідно клієнту не потрібно знати адресу та порт UAA серверу, а

використовувати порт API Gateway та шлях, згенерований за описаним вище правилом.

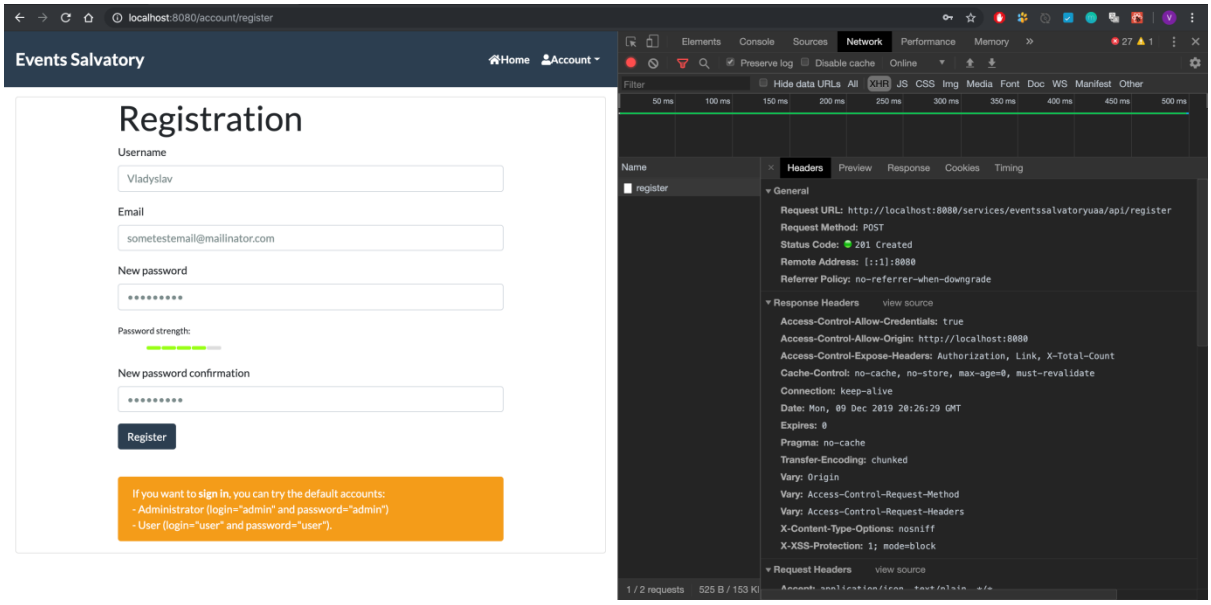


Рисунок 3.20 – Заповнена форма реєстрації та запит до UAA серверу

Власне можливо виконати запит і напряму, щоб продемонструвати це, використаймо застосунок Postman, який допомагає виконувати запити до певних API(рисунок 3.21). Postman це API клієнт який дозволяє робити запит до розроблюваного API. І це дуже зручно адже для розробки та тестування розроблюваного API не потрібно створювати клієнтську частину додатку, а достатньо всього виконувати запити використовуючи простий на гнучкий інтерфейс Postman API.

Для того щоб виконати запит до UAA серверу напряму нам необхідно додати у тіло запиту такі поля:

- email – email адреса користувача;
- langKey – мова клієнтської частини;
- login – ім’я користувача;
- password – пароль користувача.

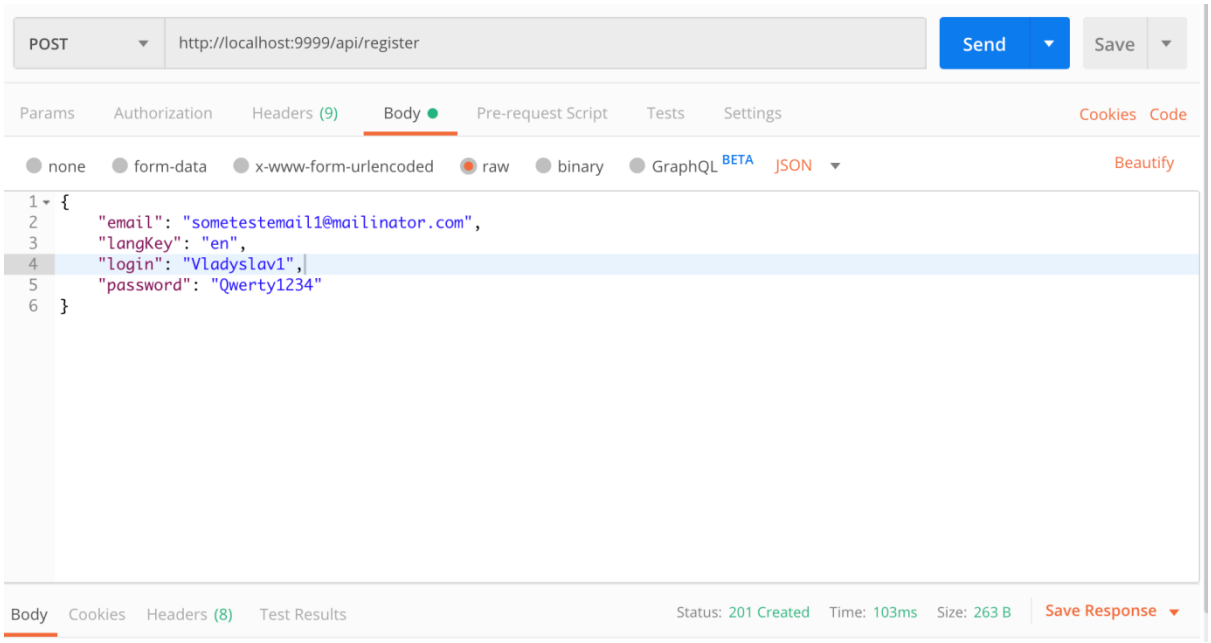


Рисунок 3.21 – Прямий запит на реєстрації до UAA серверу

Після цього можемо зайти в систему за адміністратора та переконатись, що обидва користувача були зареєстровані(рисунок 3.22).

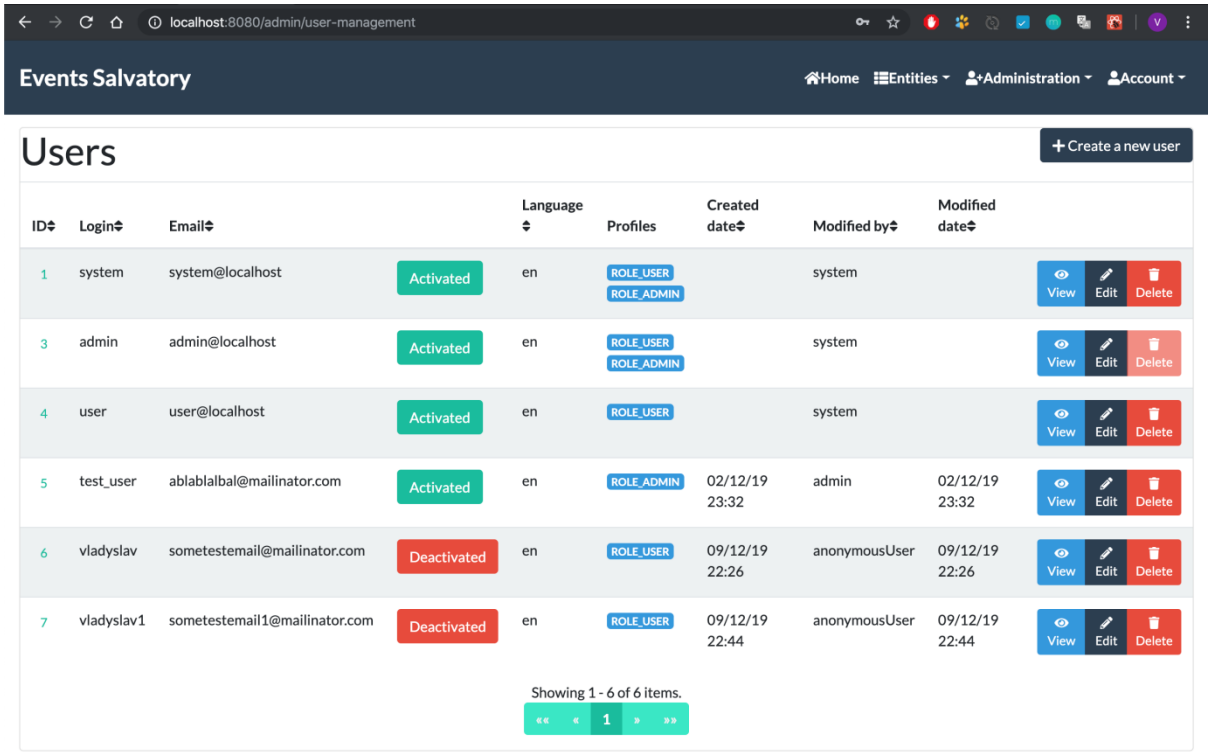


Рисунок 3.22 – Панель адміністратора для управління користувачами

Після процесу реєстрації розглянемо процес авторизації у системі. На інтерфейсі користувача буде доступна кнопка Account. Після того як

користувач натисне на неї, йому буде наданий вибір, зареєструвати нового користувача чи ввійти в систему уже існуючим. Цього разу необхідно обрати ввійти існуючим.

Після цих дій користувачу відкриється форма логіну(рисунок 3.23), де йому буде необхідно ввести своє ім'я та пароль. Після того як користувач увів свої дані у форму логіну та натиснув на кнопку login, буде зроблений запит, по вже описаній вище схемі. Якщо користувач не пройшов автентифікацію(заданий обліковий запис не існує або пароль невірний), то користувачу буде показано вікно з помилкою.

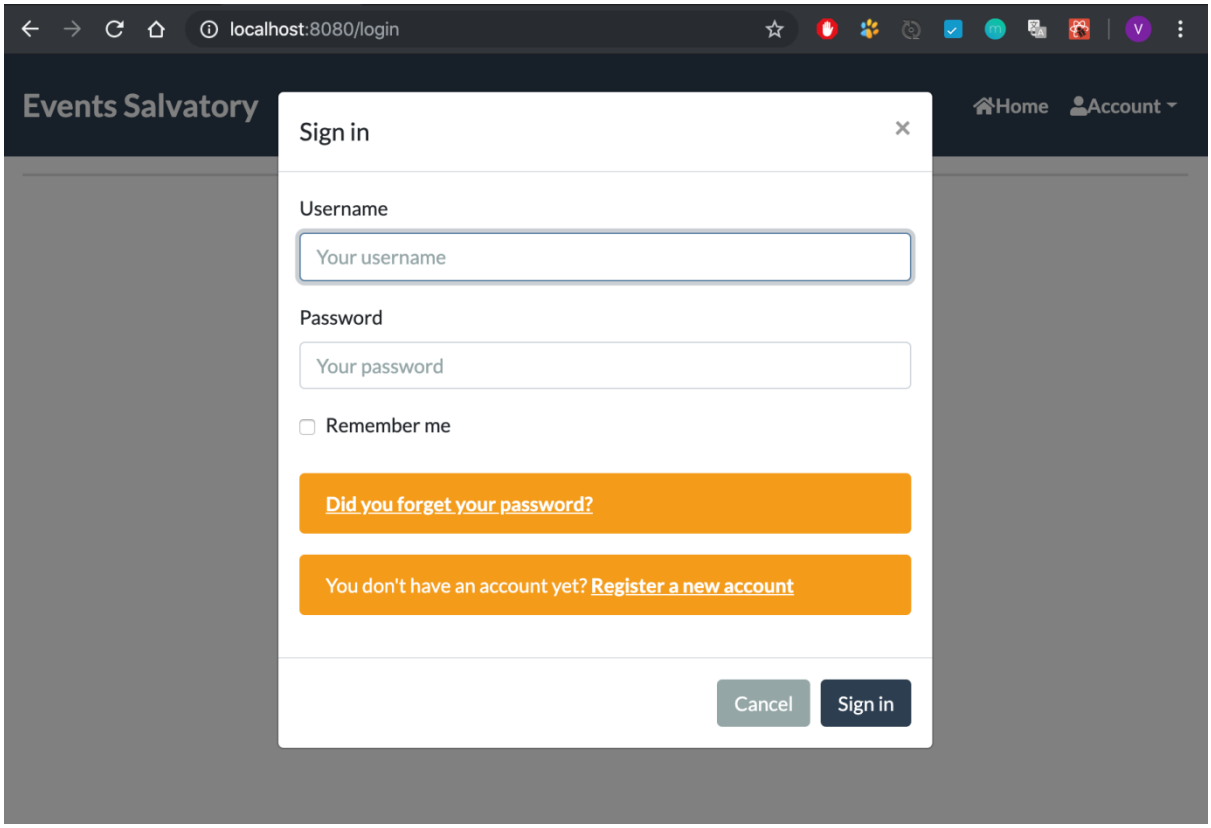


Рисунок 3.23 – Форма логіну

Якщо є користувач успішно пройшов автентифікацію, то UAA сервер згенерує JWT токен(рисунок 3.24) для користувача та збереже його до сховища cookie. JWT токен буде зберігати дані про користувача, такі як його ім’я та права доступу.

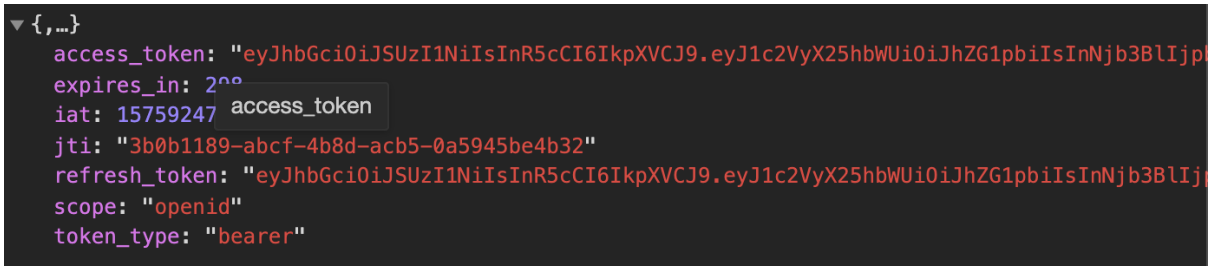


Рисунок 3.24 – Відповідь на запит авторизації, що містить JWT токен

Далі, щоб мати можливість доступу до ресурсів, таких як списку подій чи організацій, в запиті потрібно передавати даний токен в запиті. На сервері він буде проходити валідацію, декодуватись та у разі наявності відповідних прав, буде повернута відповідь клієнту зробившому запит. Для прикладу наведемо запит, що має повернути список усіх подій(рисунок 3.25).

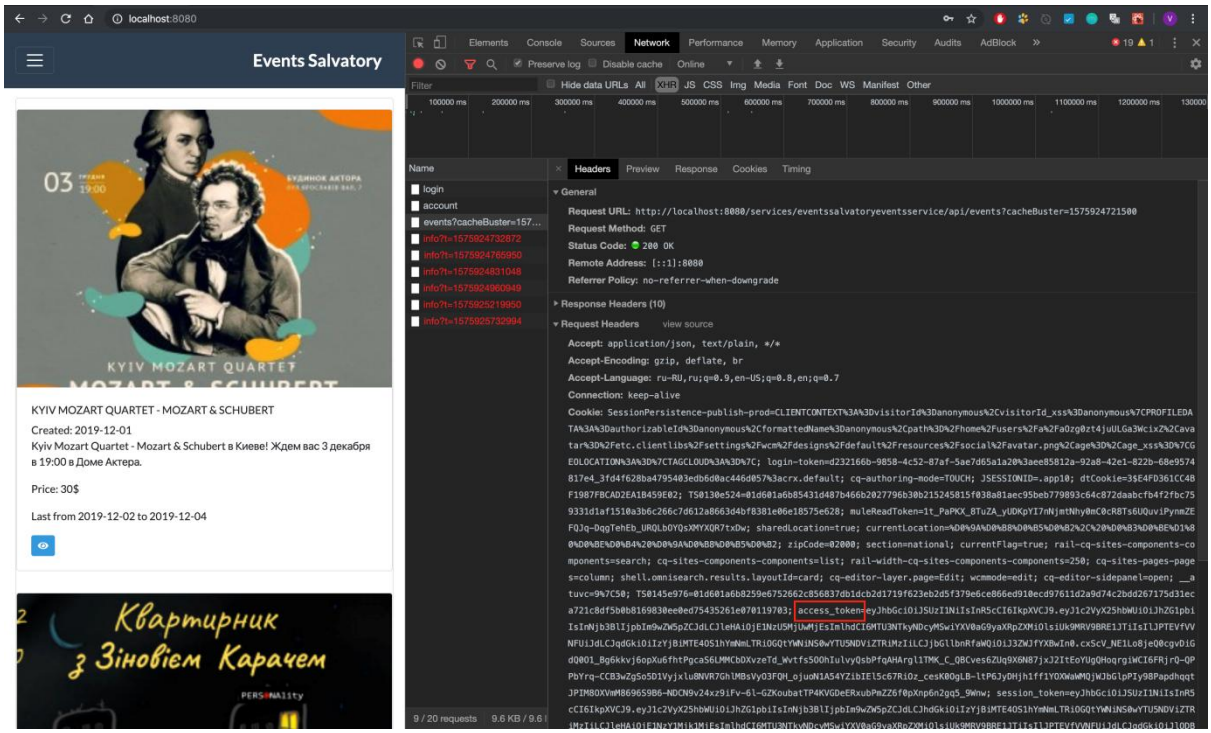


Рисунок 3.25 – Домашня сторінка системи та запит до Events мікросервісу

У відповідь на запит отримаємо JSON масив з подіями(рисунок 3.26). JSON масив представляє собою впорядкований набір об’єктів.

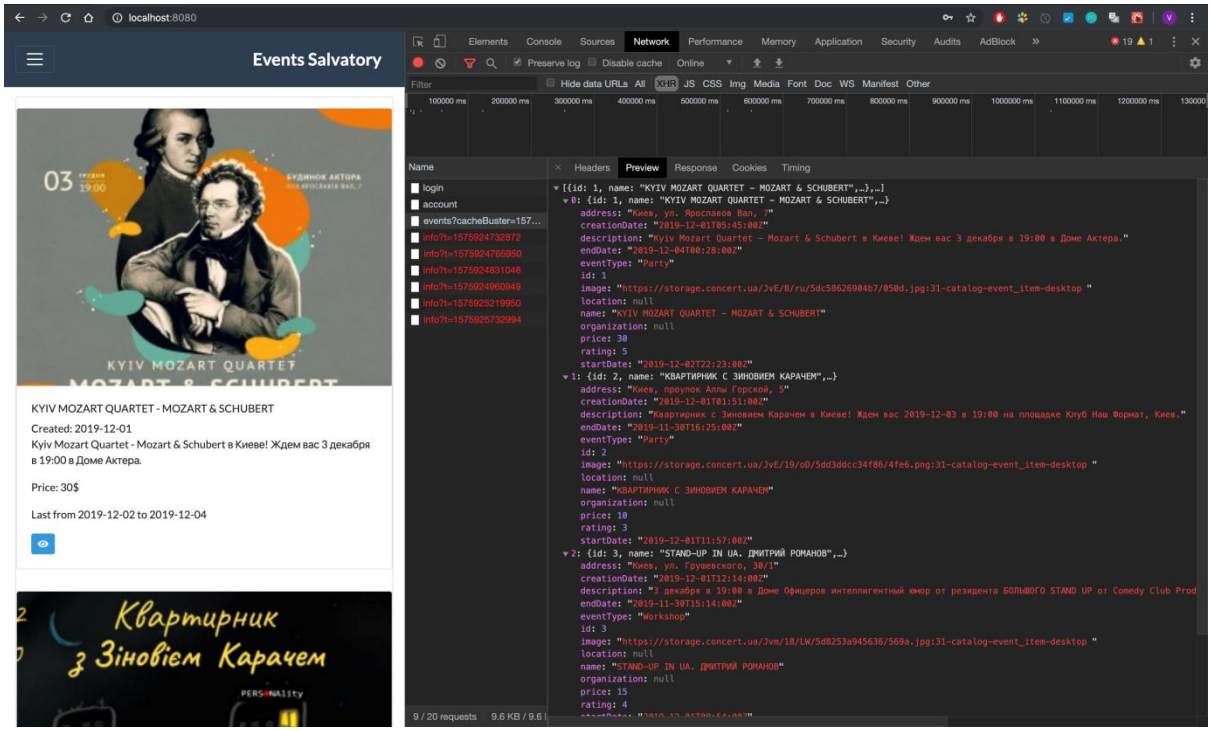


Рисунок 3.26 – Домашня сторінка та відповідь Events мікросервісу на запит

Адміністратор також має можливість переглядати(рисунок 3.27) та модифікувати існуючі події(рисунок 3.28), а також створювати нові.

Рисунок 3.27 – Сторінка події

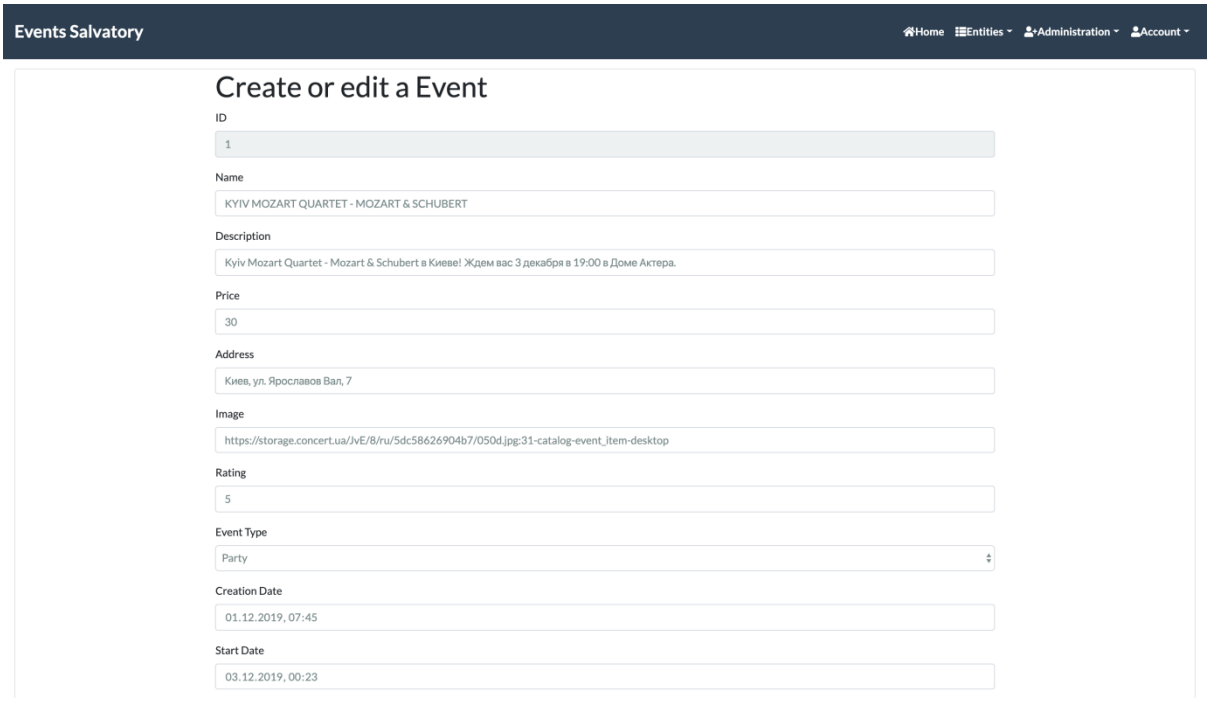


Рисунок 3.28 – Сторінка модифікації події

Registry сервер також має вбудований графічний інтерфейс і ми можемо за його допомогою переглянути, для прикладу, список підключених до серверу мікросервісів(рисунок 3.29).

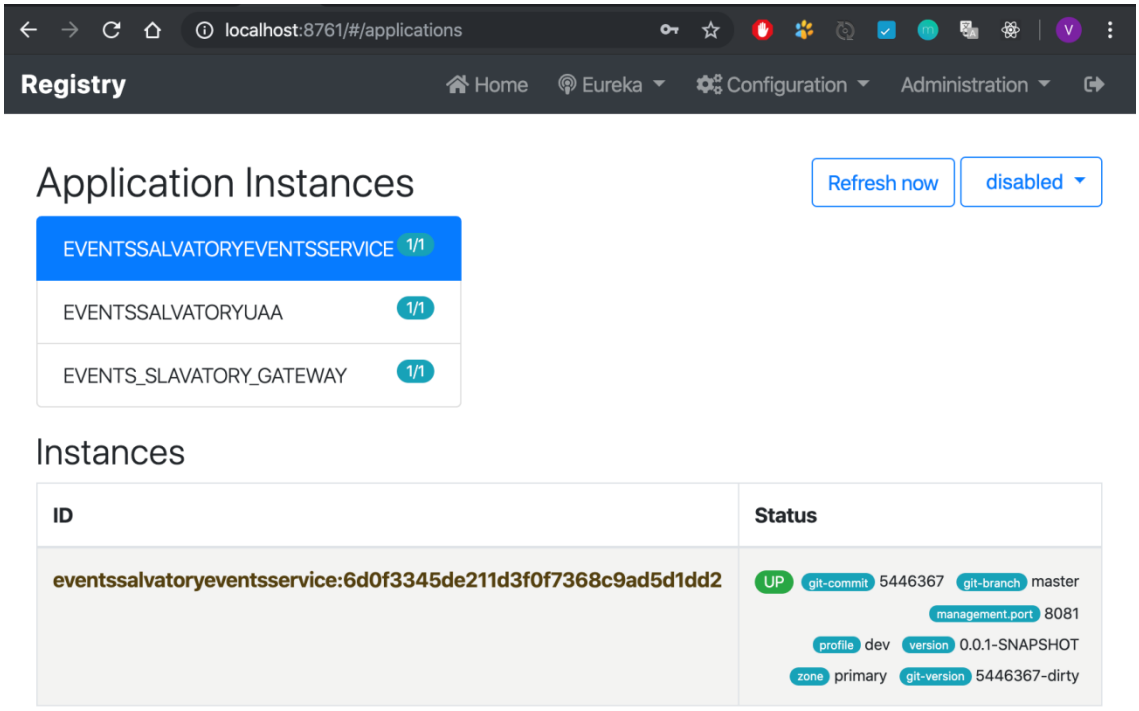


Рисунок 3.29 – Інтерфейс Registry серверу для перегляду підключених мікросервісів

На рисунку можемо побачити, що в даний момент часу Registry сервер працює з трьома мікросервісами.

3.6 Аналіз розробленого рішення

Розроблена вебсистема являється системою побудованою на основі мікросервісів. До її переваг можна віднести наведені нижче особливості.

– Легка масштабованість.

Розроблена система має великий потенціал у масштабуванні. Розроблений прототип являє собою необхідний каркас для роботи з необмеженою кількостю мікросервіссів, адже він виконує такі ключові функції вебсистеми як авторизація та автентифікація користувача в системі, робота з конфігураційними файлами, балансування навантаження, знаходження та пов’язування незалежних сервісів, тощо. Тому для створення та додавання нового мікросервісу до системи потребує мінімум додаткових зусиль.

– Гнучкість.

Розроблювана система є надзвичайно гнучкою та дозволяє використовувати різноманітні технології та бази даних для створення нових сервісів.

– Зручність використання.

Розроблені мікросервіси являються автономними, тобто їх можливо використовувати окремо як сервіси, що надають свої відкриті API. Скажемо UAA сервер можливо використовувати і в інших проектах, просто необхідно робити запити до визначених в сервісі кінцевих точок і сервер все так же буде генерувати JWT токен та надавати та контролювати інформацію про користувача.

– Відмовостійкість на безпроблемне розгортання.

Розроблене рішення є доволі відмовостійким і, скажемо, відмова Events сервісу не призведе до відмови всієї системи і користувальницький

інтерфейс, як і авторизація та автентифікація для прикладу, все ще буде працювати справно. Також процес розгортання кожного сервісу є доволі простим та зручним і не вплине на роботу усієї системи, до того ж при внесенні змін до певного сервісу, немає необхідності перезавантажувати усю систему, а достатньо лише перерозгорнути необхідний мікросервіс.

Підсумовуючи варто зауважити, що розроблене рішення являється готовим шаблоном для розробки нових систем на основі мікросервісів. API Gateway сервер, UAA сервер та Registry сервер є повноцінним та необхідним мінімумом для розробки нових мікросервісних систем. Необхідно лише додавати нові мікросервіси з новою логікою, беручи за шаблон Events сервіс. Усі перераховані плюси роблять безкомпромісним вибір мікросервісної архітектури з поміж монолітної та сервісно-орієнтованої, які не мають описаних переваг, а сам прототип допомагає вирішити головний недолік мікросервісної архітектури – її складність, що дозволить розробникам відразу зосередитись на розробці бізнес-логіки і уникнути неприємностей на старті.

3.7 Висновок до розділу

В даному розділі було розроблене програмне рішення на основі вимог поставлених у попередньому розділі. Було розроблено чотири мікросервіси, що реалізували більшість з шаблонів описаних у другому розділі. Було детально розглянуто їх реалізації. Кожен з сервісів має модуль даних, модуль бізнес логіки, модуль конфігурацій, модуль доступу до даних та модуль безпеки. Для проекту API Gateway був розроблений інтерфейс користувача для тестування розробленої системи.

4 РОЗРОБКА БІЗНЕС-ПЛАНУ ПРОЕКТУ

В даному розділі буде наведено розробку стартап-проекту за темою роботи. Буде проведений аналіз можливого впровадження проекту.

4.1 Опис ідеї стартап проекту

Ідея проекту розробити систему по пошуку подій на основі мікросервісної архітектури. Основною ціллю є саме розробка вдалої архітектури, що буде привабливою для клієнта, що міг би замовити розробку подібної системи.

Окремо розглянемо напрямки застосування ідеї та її корисність для користувача у таблиці 4.1 та для замовника у таблиці 4.2.

Таблиця 4.1 – Опис ідеї стартап-проекту для користувача

Зміст ідеї	**Напрямки застосування**	**Вигоди користувача**
Створити систему для пошуку подій з використанням мікросервісної архітектури, що дозволить користувачу швидко та зручно шукати різноманітні події	Веб-система для пошуку подій	Відсутність необхідності витрачати купу часу на пошук необхідної інформації на величезній кількості сторонніх сервісів
		Зручність. Вся інформація про події знаходиться в одному місці та має зручний користувацький інтерфейс
		Інтеграція з платіжними системами надає змогу не тільки переглядати інформацію про події, а й відразу придбати квитки при необхідності

Таблиця 4.2 – Опис ідеї стартап-проекту для замовника

Зміст ідеї	**Напрямки застосування**	**Вигоди замовника**
Розробити систему, яка буде гнучкою та легко масштабованою. Створена система має мати можливість працювати з різноманітними стеками технологій.	Розробка нової високонавантаженої системи	Простота у масштабуванні системи при необхіності
		Гнучкість при наймі розробників
		Можливість системи швидко адаптуватись під нові технологій/фреймворки

Після огляду усіх конкурентних технологій, було виявлено, що головними конкурентами являються SOA та Монолітні застосунки. Перелік сильних та слабких сторін розроблюваної архітектури дозволить краще зрозуміти її місце у порівнянні з конкурентами. Визначення сильних та слабких сторін проекту наведено у таблиці 4.3.

Таблиця 4.3 – Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№	**Техніко-економічні характеристики ідеї**	**Потенційні товари/концепції конкурентів**			**W**	**N**	**S**
		Мій проект (MSA)	Конкурент 1 SOA	Конкурент 2 Моноліти			

Продовження таблиці 4.3

1	Дизайн архітектури	Сервіси побудовані як окремі, незалежні компоненти цілої системи, що орієнтовані на бізнес потреби.	Сервіси можуть бути як невеличкого розміру так і бути великими сервісами, що вирішують безліч бізнес завдань.	Монолітний застосунок являється одним великим сервісом і доволі часто розростаєтьс я до таких розмірів, що розуміти його або масштабуват и стає досить тяжко		+	
2	Практична корисність	Сервіси надають певне API, що дає доступ до реалізації тих бізнес потреб які реалізує кожен сервіс.	До сервісів можливо отримати доступ через такий стандартний протокол передачі даних	Монолітний застосунок являється єдиним сервісом, тому перевикорис тання		+	

			як SOAP, через SOA Bus.	функціональних частин			

Продовження таблиці 4.3

2				монолітного застосунку є лімітованим			
3	Легка масштабованість	Сервіси існують як незалежні компоненти для розгортання, тому їх легко змінювати та масштабувати без значного впливу на систему в цілому.	Сервіси мають певну зв’язаність, що призводить до певних проблем у масштабуванні.	Оскільки моноліт це єдиний застосунок, його масштабування може бути справжнім викликом.			+
4	Гнучкість у виборі технологій	Оскільки кожен сервіс являється автономним, розробники мають можливість використовувати різні стеки технологій для різних	Можливість використовувати різні технології присутня, але наявність рівня агрегації SOA Bus накладає певні обмеження.	Монолітний застосунок має використовувати лише єдиний стек технологій.			+

		мікросервісів.					

Продовження таблиці 4.3

5	Гнучкість у побудові команд розробників	Існує можливість створення різноманітних команд, що будуть працювати над різними сервісами та з різними технологіями, що надає велику гнучкість у наймі.	+	-			+
6	Відмовостій кість системи	Кожен компонент є автономним сервісом, тому його відмова не зумовить відмову всієї системи. Тим паче, що частіше за все розгортається декілька екземплярів	Середня	Низька. Поломка будь-якого компоненту скоріш за все потягне за собою збій всієї системи.			+

		сервісу, тому при поломці одного, система					

Продовження таблиці 4.3

6		перерозподіляє навантаження по решті сервісів того ж типу.					
7	Простота освоєння	Розробка на основі мікросервісної архітектури є досить важким процесом, адже потрібно налаштувати та розробити величезну кількість сервісів, але якщо допустити, що окремі команди працюють над окремими сервісами, то невеликі компоненти системі є простими для розуміння, тоді як	Середня	Середня/Проста		+	

		система в цілому, все ж таки досить тяжка.					

Продовження таблиці 4.3

8	Ремонтопридатність	Оскільки кожен сервіс є автономним, то поломка окремого сервісу не зламає систему в цілому. Також є можливість окремо ремонтувати сервіс та перерозгортати його не сильно впливаючи на систему в цілому.	Середня	Середня			+
9	Захищеність	Захист по протоколу Oauth2 та за допомогою JWT токену.	Захист по протоколу Oauth2 та за допомогою JWT токену.	Захист по протоколу Oauth2 та за допомогою JWT токену.		+	

4.2 Технологічний аудит проекту

Далі проведемо технологічний аудит проекту, в якому розглянемо, які технології будуть використатися при розробці даного проекту. Аудит

слугує для того, щоб проаналізувати список обраних технологій та зробити висновок на доцільності їх використання. Результати наведені в таблиці 4.4.

Таблиця 4.4 – Технологічний аудит проекту

Зміст ідеї	**Технології її реалізації**	**Наявність технологій**	**Доступність технологій**
Система пошуку подій побудована на основі мікросервісної архітектури	Spring stack Netflix OSS stack	Технології наявні та добре працюють разом. Необхідно правильно їх об’єднати між собою	Технології доступні
	Spring Boot	Технологія наявна.	Технологія доступна
	Spring Cloud	Технологія наявна	Технологія доступна
	Spring Data	Технологія наявна	Технологія доступна
	Spring Cloud Security	Технологія наявна	Технологія доступна
	Netflix Eureka	Технологія наявна. Відкрите ПЗ	Технологія доступна

	Netflix Zuul	Технологія наявна. Відкрите ПЗ	Технологія доступна
	Netflix Hystix	Технологія наявна. Відкрите ПЗ	Технологія доступна

Продовження таблиці 4.4

	Netflix Ribbon	Технологія наявна. Відкрите ПЗ	Технологія доступна
	Maven	Технологія наявна	Технологія доступна
	Webpack	Технологія наявна	Технологія доступна
	React.js	Технологія наявна	Технологія доступна
Обрані технології: Spring Framework stack, Netflix OSS stack, JS/React.js stack			

Технологічний аудит проекту показав, що для розробки додатку на основі мікросервісної архітектури досить привабливою є мова Java там фреймворк Spring, що має багато модулів для розробки саме мікросервісів та інтеграцію з бібліотеками Netflix OSS.

4.3 Аналіз ринкових можливостей запуску

Перед виходом на ринок потрібно проаналізувати чи буде попит на даний проект. До того ж існує досить велика кількість загроз на ринку для успішного запуску будь-якого проекту. Проведемо попередню характеристику ринку у таблиці 4.5.

Таблиця 4.5 – Попередня характеристика потенційного ринку стартап-проекту

№ п/п	**Показники стану ринку (найменування)**	**Характеристика**
1	Кількість головних гравців, од	Безліч
2	Загальний обсяг продаж, грн/ум.од	–
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень)	Велика кількість систем, які вже успішно працюють в конкретних сферах. Складність реалізації на старті.
5	Специфічні вимоги до стандартизації та сертифікації	Відсутні
6	Середня норма рентабельності в галузі (або по ринку), %	~80%

Після аналізу потенційного ринку стає зрозуміло, що на ринку існує велика кількість конкурентів, що пропонують різні рішення для розробки проектів. Тому потрібно добре зрозуміти хто є потенційними клієнтами

стартапу та визначити цільову групу. Аналіз потенційних клієнтів стартап-проекту наведений у таблиці 4.6.

Таблиця 4.6 - Характеристика потенційних клієнтів стартап-проекту

№ п/п	**Потреба, що формує ринок**	**Цільова аудиторія (цільові ринки)**	**Відмінності у поведінці різних цільових груп клієнтів**	**Вимоги споживачів до товару**
1	Потреба мати легко масштабовану та максимально гнучку систему, що дозволить замовнику швидко реагувати на зміни	Будь-які установи/бізнеси/ринки	Система має бути легко масштабованою та гнучкою. Система повинна надавати можливість розробляти її на різноманітних технологіях, бути надійною, відмовостікою та захищеною. Поріг входу для нових розробників має бути досить низьким.	Система має надавати привабливий та зручний клієнтський інтерфейс. Бути зручною та швидкою.

Після аналізу потенційних клієнтів стартап-проекту проведемо аналіз факторів загроз та факторів можливостей у таблицях 4.7 та 4.8.

Таблиця 4.7 – Фактор загроз

№	**Фактор**	**Зміст загрози**	**Можлива реакція**

п/п			**компанії**
1	Складність реалізації	Мікросервіси хоч і надають величезну кількість переваг, та це	Спрощення складності системи шляхом

Продовження таблиці 4.7

1		випливає у досить велику складність при розробці	комбінації та з’єднання сервісів в більші.
2	Відсутність зацікавленості у нових замовників	Деякі замовники можуть вирішити, що система побудована на основі мікросервісної не відповідає їх вимогам	Покращення процесу розробки систему та підвищення якості розроблюваних рішень шляхом постійної взаємодії з замовниками
3	Зміна трендів	Мікросервісна архітектура може застаріти	Гнучкість архітектурного підходу дозволяє досить просто адаптувати систему до нових технологій
4	Критичні помилки	Виявлення помилок в роботі системи	Своєчасний випуск оновлень, надання вичерпної консультації службою підтримки

Таблиця 4.8 – Фактор можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Зацікавленість інвесторів	Можливість отримати інвестиції	Збільшення штату розробників

Продовження таблиці 4.8

2	Інтеграція з іншими сервісами та плафтормами	Розширення потенційної кількості користувачів та рекламодавців. Також гнучкість у виборі технології при розробці	Збільшення кількості персоналу, що працює над інтеграційними питаннями
3	Співпраця з іншими компаніями	Поєднання данних з різноманітних сервісів пошуку заходів за допомогою відкритих API	Розширення функціоналу системи
4	Співпраця з фізичними особами	Залучити користувачів до популяризації системи	Підвищення рівня вдоволеності користувачів

Далі проведемо аналіз конкуренції на ринку у таблиці 4.9.

Таблиця 4.9 - Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	**В чому полягає дана характеристика**	**Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)**

1. Вказати тип конкуренції - чиста	Велика кількість незалежних гравців в окремих сферах на ринку	Пропозиція власного підходу до розробки системи, її спрощення, що дозволить пришвидшити розробку та зменшити витрати
2. За рівнем конкурентної боротьби	Основними конкурентами системи є веб-системи по	Пропозиція кращої системи за менші кошти.

Продовження таблиці 4.9

2. За рівнем конкурентної боротьби - національний	пошуку подій в рамках країни.	
3. За галузевою ознакою - внутрішньогалузева	Співпраця з підприємствами та установами, робота яких пов’язана з подіями	Запропонувати спрощений процес інтеграції з сервісами інших компаній
4. Конкуренція за видами товарів: - товарно-видова.	Велика кількість існуючих систем в окремих галузях	Розширення функціоналу системи для вирішення потреб кожної конкретної компанії
5. За характером конкурентних переваг: - цінова.	Замовник платить за розробку та використання системи	Орієнтація на потребу замовників

Далі варто провести аналіз конкуренції у галузі методом п’яти сил Майкла Портера.

Таблиця 4.10 - Аналіз конкуренції в галузі за М. Портером

Складові аналізу:	**Прямі конкуренти**	**Потенційні конкуренти**	**Постачальники**	**Клієнти**	**Товари-замінники**
	Інші компанії по розробці програмного	Невелкі компанії по розробці ПЗ. Фрілансери.	Доступ до системи постачається за рахунок	Будь-які підприємства, що бажають замовити	Кожен з продуктів конкурентів є, частково,

Продовження таблиці 4.10

Складові аналізу:	забезпечення, що є лідерами на ринку розробки ПЗ та до яких уже є довіра в Україні: EPAM, Ciklum, Global Logic, Data Art.		мережі Інтернет	розробку системи	замінником
Висновки:	Існує досить велика кількість конкурентних компаній	Вихід на ринок залежить від умов замовника,	Не впливають ніяк на те що відбувається на	Оскільки існує досить велика кількість конкурентів,	Кожна система розроблена компанією-конкуренто

	різних розмірів.	конкретної ситуації на ринку праці та кількості конкурентних рішень.	ринку	основною ціллю є максимальне задоволення потреб клієнта за оптимальною ціною	м на основі будь-якої архітектури є потенційним замінником

Далі варто навести фактори, що роблять проект конкуренто спроможним, це зробимо у таблиці 4.11.

Таблиця 4.11 - Обґрунтування факторів конкурентоспроможності

№ п/п	**Фактор конкурентоспроможності**	**Обгрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущих)**
1	Гнучкість архітектурного рішення	Розробка системи з використанням мікросервісної архітектури робить систему гнучкою та легкою для масштабування та модифікації, що є привабливим для замовників, що планують масштабувати своє підприємство
2	Цінова політика	Приваблива цінова політика для досить складної системи привабить додаткових клієнтів
3	Відмовостійкість	Архітектура системи передбачає досить

		високу відмовостійкість, що дозволить замовнику буди впевненим, що збій системи не принесе йому ніяких додаткових витрат, так як збій системи сам по собі фактично не можливий
4	Оптимізація	Розроблюваний застосунок буде просто оптимізувати та розширяти

Продовження таблиці 4.11

5	Універсальність	Запропоноване архітектурне рішення може бути використане для будь-яких замовлень

Таблиця 4.12 - Порівняльний аналіз сильних та слабких сторін

№ п/п	**Фактор конкурентоспроможності**	**Бали 1-20**	**Рейтинг товарів-конкурентів у порівнянні з моїм продуктом**						
			-3	-2	-1	0	+1	+2	+3
1	Гнучкість архітектурного рішення	18			+				
2	Цінова політика	20		+					
3	Відмовостійкість	15				+			
4	Оптимізація	17				+			
5	Універсальність	18			+				

Далі можливо виокремити усі сильні та слабкі сторони проекту, це зроблено у таблиці 4.13 за допомогою SWOT аналізу.

Таблиця 4.13 - SWOT- аналіз стартап-проекту

Сильні сторони:	**Слабкі сторони:**
Сильними сторонами запропонованого архітектурного рішення є гнучкість, легка масштабованість, універсальність для використання. Великою перевагою є ціна на створення ПЗ з використанням обраного архітектурного рішення у порівняння з ринком.	Висока конкуренція Важкий процес розробки Складний процес розгортання

Продовження таблиці 4.13

Можливості:	**Загрози:**
Можливостями є розширення складу команди. Створення щаблону проекту, що дозволить прискорити процес розгортання проекту. Інтеграція з різноманітними сервісами, що може пришвидшити процес розробки та спростити його. Існує можливість використання інших архітектурних рішень для залучення нових клієнтів.	Мікросервісна архітектура може застаріти або її популярність може зменшитись. Компанії конкуренти можуть запропонувати краще рішення або ціну, що може привести до відсутності попиту. Компаній конкурентів може з’явитись більше, що додасть більше конкуренції.

Таблиця 4.14 - – Альтернативи ринкового впровадження стартап-проекту

№ п/п	**Альтернатива (орієнтовний комплекс заходів) ринкової поведінки**	**Ймовірність отримання ресурсів**	**Строки реалізації**
1	Використання альтернативного архітектурного підходу до побудови системи	Середня	Декілька місяців
2	Реалізація функціоналу частково з поступовим додаванням нових сервісів	70%	Декілька місяців

Продовження таблиці 4.14

3	Використання альтернативного стеку технологій	Середня	Декілька місяців
4	Маркетингова компанія для приваблювання нових замовників. Реалізація демо проектів для зацікавлення.	80%	Декілька місяців
5	Розробка застосунку з повним функціоналом у найкоротші строки на основі розробленого проекту-шаблону	80%	Декілька місяців

4.4 Розроблення ринкової стратегії проекту

Для успішного запуску проекту необхідно знати цільові групи проекту. Наведемо їх у таблиці 4.15.

Таблиця 4.15 - Вибір цільових груп потенційних споживачів

№ п/п	**Опис профілю цільової групи потенційних клієнтів**	**Готовність споживачів сприйняти продукт**	**Орієнтовний попит в межах цільової групи (сегменту)**	**Інтенсивність конкуренції в сегменті**	**Простота входу у сегмент**

Продовження таблиці 4.15

1	Малий бізнес та стартапи	Готові	Малий	Висока	Вхід у сегмент є досить складним, адже запропонований архітектурний підхід є досить складним і краще за все підходить системам, що планують рости та масштабуватись. Розробка та підтримка таких

					систем є доволі довгим процесом, коли як для малого бізнесу та стартапів необхідне швидке рішення. Тим не менш, існує комплекс заходів для пришвидшення розробки та запуску проекта, таких як розробка системи з шаблона.

Продовження таблиці 4.15

2	Середній бізнес	Готові	Середній	Висока	Вхід у сегмент є середнім по складності, все залежить від цілей бізнесу. Якщо бізнес має намір рости та масштабуватись даний архітектурний підхід являється вдалим рішенням,

					але якщо бізнес не планує рости, то дане рішення може бути не вдалим ходом
3	Великий бізнес	Готові	Високий	Висока	Вхід у сегмент є простим, адже даний архітектурний підхід найкраще підходить для створення великих систем з купою сервісів.

Продовження таблиці 4.16

4	Державні установи	Готові	Середній/Високий	Висока	Даний сегмент частіше за все являє собою великі та складні системи тому простота входу у сегмент є середньою/великою
5	Фізичні особи	Готові	Середній	Висока	Вхід у сегмент є середньої простоти адже для фізичних

					особ скоріш за все потрібно розробляти не складні системи, але дане архітектурне рішення можливо адаптувати і для малих систем
Цільові групи: №2, №3, №4					

Наступним кроком є визначення базової стратегії розвитку продукту, що наведена у таблиці 4.16.

Таблиця 4.16 - Визначення базової стратегії розвитку

№	**Обрана альтернатива розвитку проекту**	**Стратегія охоплення ринку**	**Ключові конкурентос-проміжні позиції відповідно до обраної альтернативи**	**Базова стратегія розвитку**
1	Вихід на ринок з повним	Проведення маркетингової	Використання альтернативного	Концентрований

	функціоналом. Створення шаблону проекта, що дозволить швидше почати розробку застосунку та зможе привабить малий на середній бізнес. Створення фінансової політики, що зробить компанію привабливішою за конкурентів.	кампанії. Пропозиції різноманітних цінових планів для бізнесів різного розміру.	стеку технологій. Використання альтернативного архітектурного підходу.	маркетинг

Далі визначимо базову стратегію конкурентної поведінки, та додамо результати у таблицю 4.16.

Таблиця 4.17 - Визначення базової стратегії конкурентної поведінки

№	**Чи є проект «першопрохідцем» на ринку?**	**Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?**	**Чи буде компанія копіювати основні характеристики товару конкурента, і які?**	**Стратегія конкурентної поведінки**

1	Ні	Компанія буде як шукати нових користувачів так і відбирати існуючих у конкурентів.	Копіювання вигідних характеристик є звичайною практикою, адже це очевидний наслідок конкуренції.	Стратегія зайняття конкурентної ніші

Таблиця 4.18 - Визначення стратегії позиціонування

№	**Вимоги до товару цільової аудиторії**	**Базова стратегія розвитку**	**Ключові конкурентоспроможні позиції власного стартап-проекту**	**Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключо-вих)**

Продовження таблиці 4.18

1	Універсальність	Зростання	Позиціонування системи як такої, що можливо використовувати в різноманітних ситуаціях	Зручність Універсальність Гнучкість
2	Зручність	Зростання	Позиціонування системи як такої, що надає надзвичайну зручність клієнтам установи	

3	Функціональність	Стабілізація	Позиціонування системи як такої, що надає велику кількість функціональних можливостей клієнту	
4	Гнучкість	Зростання	Позиціонування системи як такої, що є надзвичайно гнучкої та легкої для змін	

4.5 Розроблення маркетингової програми

Таблиця 4.19 - Визначення ключових переваг концепції потенційного товару

№	**Потреба**	**Вигода, яку пропонує товар**	**Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)**
1	Універсальність	Архітектурне рішення може використовувати для рішення великої кількості різноманітних задач	Гнучкість, Масштабованість, Відмовостійкість.

Продовження таблиці 4.19

2	Гнучкість	Система є гнучкою для модифікації та масштабування	Систему можливо швидко масштабувати, модифікувати та інтегрувати з іншими системами
3	Дешевизн	Не дивлячись на те що	Компанія зможе запропонувати

	а	розроблювана система буде доволі складної з точки зору архітектури, цінова політика буде залишатись досить привабливою.	свої послуги досить широкій аудиторії

Таблиця 4.20 - Опис трьох рівнів моделі товару

Рівні товару	**Сутність та складові**	
I. Товар за задумом	За задумом маємо систему по пошуку подій побудовану на основі мікросервіної архітектури	
II. Товар у реальному виконанні	Властивості / характеристики:	Розмір сирцевих файлів
	Серверний модуль Клієнтський модуль	2,7Мб 1.8Мб
	Якість: тестування за допомогою юніт тестів під час розробки	
	Марка: назва розробника та назва системи	
III. Товар із підкріпленням	До продажу: Стандартна система	
	Після продажу: додаткова підтримка та можливість розширення	

Таблиця 4.21 - Визначення меж встановлення ціни

№	**Рівень цін на товари-замінники**	**Рівень цін на товари аналоги**	**Рівень доходів цільової групи споживачів**	**Верхня та нижня межі встановлення ціни на товар/послугу**

1	$4000-$100000	$4000-$100000	>$200000	$0-$10000

Таблиця 4.22 – Формування системи збуту

№	**Специфіка закупівельної поведінки цільових клієнтів**	**Функції збуту, які має виконувати постачальник товару**	**Глибина каналу збуту**	**Оптимальна система збуту**
1	Оплата послуг по розробці та підтримці програмного забезпечення	Підписання контрактів та продаж за зазначеною ціною	Дворівневий	Через сайт компанії виробника

Таблиця 4.23 – Концепція маркетингових комунікацій

№	**Специфіка поведінки цільових клієнтів**	**Канали комунікацій, якими користуються цільові клієнти**	**Ключові позиції, обрані для позиціонування**	**Завдання рекламного повідомлення**	**Концепція рекламного звернення**

Продовження таблиці 4.23

1	Заключення контракту на певний	Директ-маркетинг, інтернет сайт	Архітектурний підхід позиціонує себе як	Повідомити потенційних клієнтів про можливі	Рекламне звернення спрямовано до потенційних

	строк		альтернатива у підходах до розробки програмного забезпечення.	варіанти розробки ПЗ, показати усі позитивні риси запропонованого рішення та проінформувати про привабливу цінову політику.	клієнтів, де показуються плюси користування системою

4.6 Висновок до розділу

У даному розділі був виконаний аналіз стартап-проекту, описано його основу ідею, проведений технологічний аудит, проведено аналіз загроз та можливостей та виділено слабкі та сильні сторони.

Також було досліджено, що існує досить багато конкурентів на ринку тому необхідно надавати кращі послуги, за меншими цінами ніж у конкурентів.

Були оцінені цільові групи та клієнти стартап-проекту і було зроблено висновок, що обрана стратегія розвитку є найоптимальнішою. Факторами, що можуть вплинути на впровадження стартапу є можливість появлення нового тренду на ринку розробки ПЗ та складність побудови системи, та були запропоновані рішення даних проблем та варіанти реакцій на них.

Підсумовуючи можна сказати, що не дивлячись на велику кількість конкурентів на складність реалізації, при достатній підготовці проект є доволі перспективним.

ВИСНОВОК

У ході виконання даної магістерської дисертації було проаналізовані архітектурні підходи до розробки програмного забезпечення, виділені переваги та недоліки кожного. Після чого були зроблені висновки щодо доцільності використання мікросервісної архітектури в цілому. У зв’язку

зі знайденими недоліками мікросервісів було запропоновано використовувати низку шаблонів, які б усунули ці недоліки.

Після чого було детально проаналізовано список шаблонів, які можливо використовувати при розробці системи на основі мікросервісної архітектури. Далі був проведений аналіз та вибір технологічного стеку для створення прототипу систему для демонстрації реалізації системи з використанням оглянутих шаблонів. Також була розроблена діаграма акторів системи, що дозволило зрозуміти як саме необхідно декомпозувати систему. Все це дозволило розробити оптимальну систему, яка реалізовувала усі переваги мікросервісної архітектури та не мала б багатьох недоліків.

Наступним пунктом була розробка системи на основі прийнятих рішень. Було розроблено чотири сервіси, кожен з яких реалізовував певні шаблони та виконував свої важливі ключові функції, такі як скажем авторизація та автентифікація, надання API доступу до даних чи надання єдиної точки доступу до системи. Кожен розроблений сервіс має модуль даних, модуль бізнес логіки, модуль конфігурацій, модуль доступу до даних та модуль безпеки, кожен з яких виконує свою функцію, але вже в межах конкретного мікросервісу.

Також був проведений аналіз стартап проекту, що показав необхідність створення шаблону проекту для спрощення процесу розробки мікросервісної системи у зв’язку з великою кількістю конкурентів та складністю розробки в цілому, якщо не використовувати шаблони описані в другому розділі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Рудольф Мачадо «Monolithic Architecture» / Рудольф Мачадо – К. : 1995.

2) Томас Ерл «SOA Principles of Service Design» / Томас Ерл – К. : «Prentice», 2008

3) Microservices [Електронний ресурс] // MartinFlower – Режим доступу до ресурсу: https://martinfowler.com/articles/microservices.html

4) Pattern: API Gateway/Backend for Frontends [Електронний ресурс] – Режим доступу до ресурсу: https://microservices.io/patterns/apigateway.html

5) Circuit Breaker pattern [Електронний ресурс] // Microsoft – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker

6) Тхуан Л. Тай/Хоан К. Лам «.NET Framework Essentials: Introducing the .NET Framework» / Тхуан Л. Тай/Хоан К. Лам – К. : «O`REILLY», 2003.

7) Шилдт Г. «Java 8. Полное руководство, 9-е издание» / Шилдт Г. – К. : «Вильямс», 2015.

8) The Java Tutorials [Електронний ресурс] // Oracle – Режим доступу до ресурсу: https://docs.oracle.com/javase/tutorial/

9) Джон Карнел «Spring Microservices in Action» / Джон Карнел – К. : «MANNING», 2017.

10) Netflix OSS [Електронний ресурс] // Netflix – Режим доступу до ресурсу: https://netflix.github.io/

11) Девід А. Блек «The Well-Grounded Rubyist» . Девід А. Блек. – К. : «MANNING», 2009.

12) Nodejs Documentation [Електронний ресурс] // Nodejs – Режим доступу до ресурсу: https://nodejs.org/api/

13) Express API [Електронний ресурс] // Express – Режим доступу до ресурсу: https://expressjs.com/en/4x/api.html

14) Spring Cloud [Електронний ресурс] // Spring – Режим доступу до ресурсу: https://spring.io/projects/spring-cloud#overview

15) Spring Data JPA [Електронний ресурс] // Spring – Режим доступу до ресурсу: https://spring.io/projects/spring-data-jpa

16) Spring Cloud Netflix [Електронний ресурс] // Spring – Режим доступу до ресурсу: https://cloud.spring.io/spring-cloud-netflix/reference/html/#service-discovery-eureka-clients

17) Рагурам Баратан «Apache Maven Cookbook» » / Рагурам Баратан – К. : «PACKT», 2015.

18) JPA [Електронний ресурс] // Spring – Режим доступу до ресурсу: https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/#reference

19) Лаврентій Спілка «Spring Security in Action» / Лаврентій Спілка – К. : «MANNING», 2019.

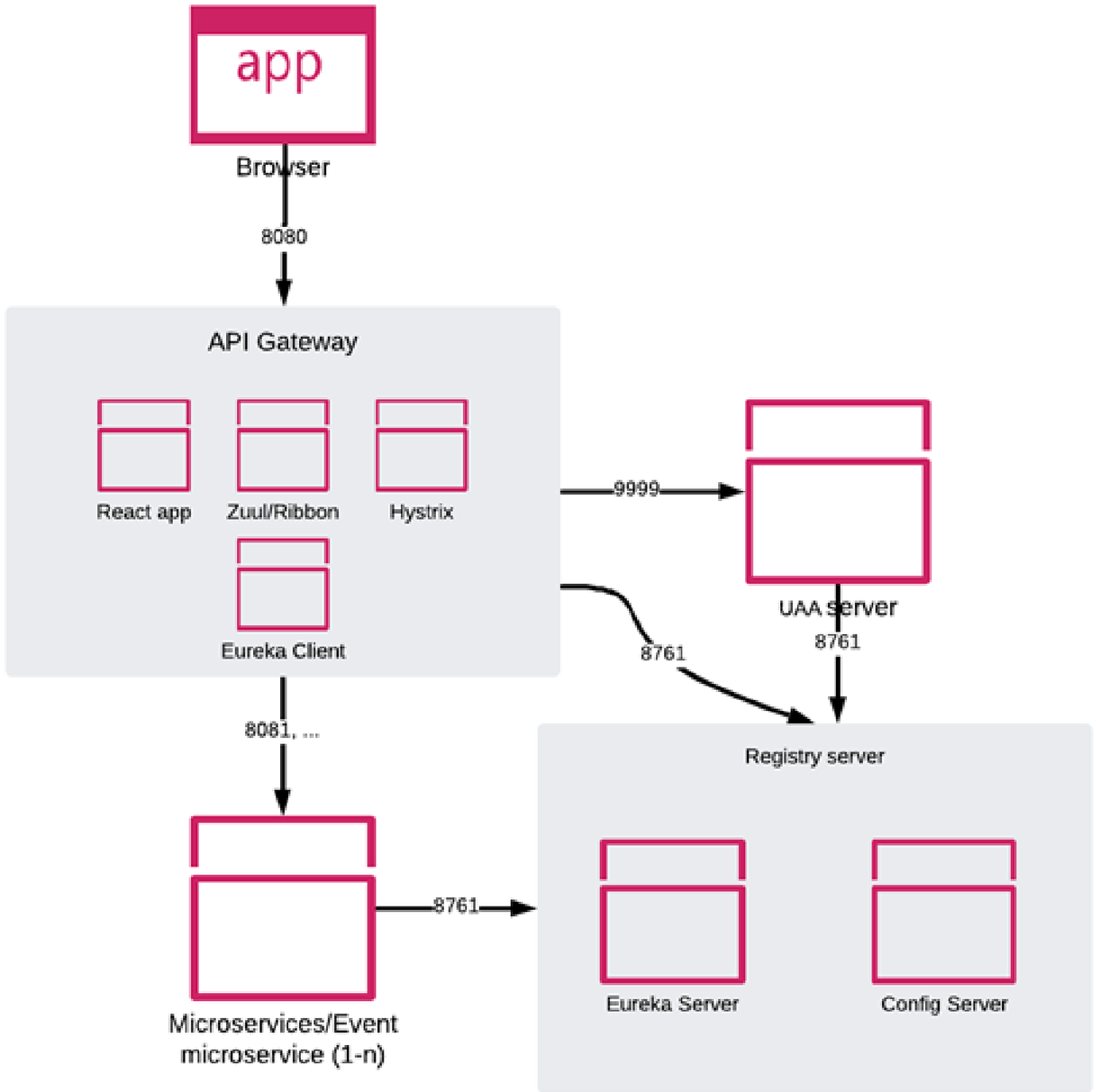
20) JSON Web Token [Електронний ресурс] // Auth0 – Режим доступу до ресурсу: https://jwt.io/introduction/

ДОДАТКИ

ДОДАТОК А

ГРАФІЧНИЙ МАТЕРІАЛ

Структурна схема розробленої системи



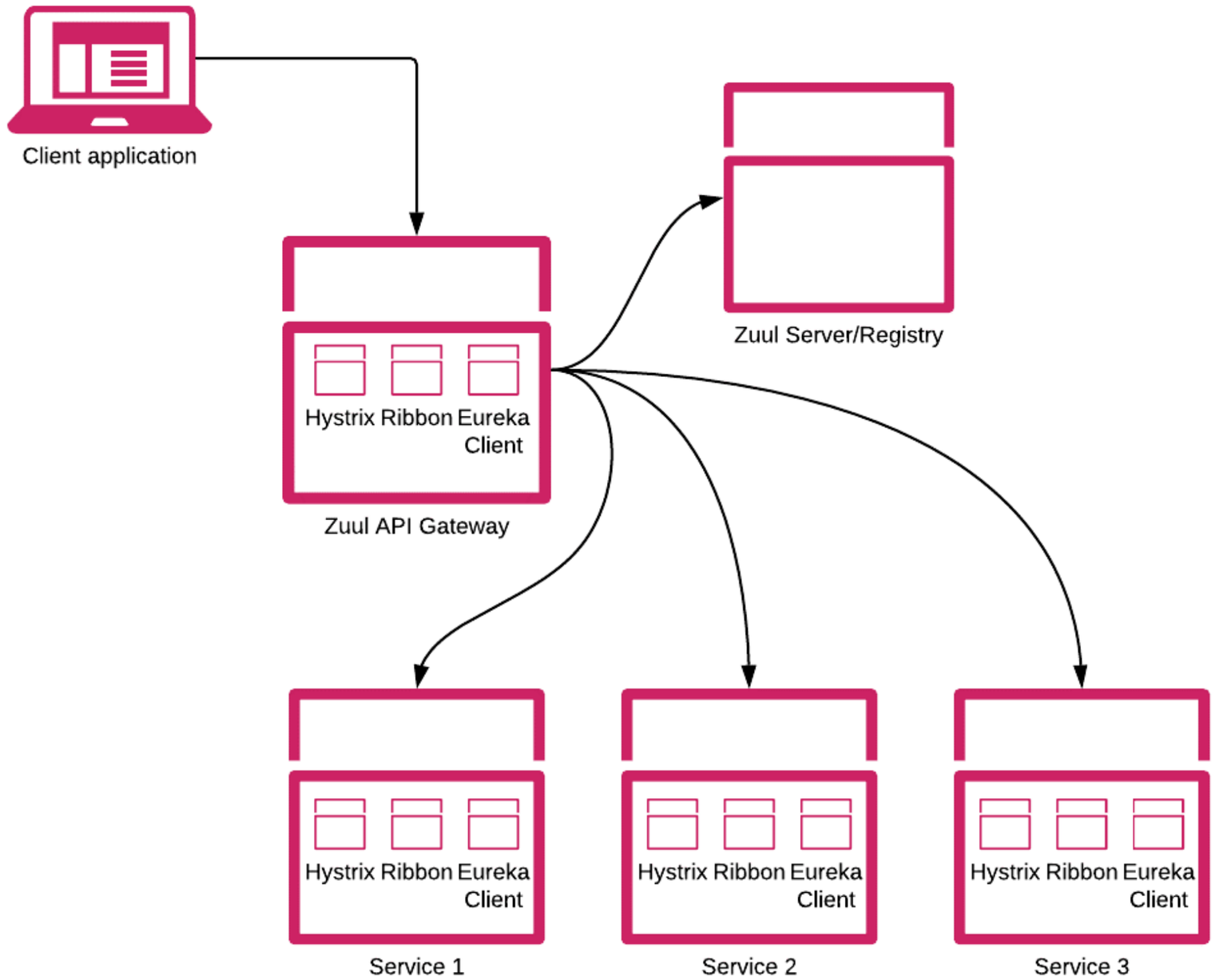
Демонстраційний плакат до магістерської дисертації

Структурна схема розробленої системи

Виконав студент гр. ІП-82мп Стеценко В.О.

Керівник Ліщук К.І.

Структурна схема API Gateway сервісу



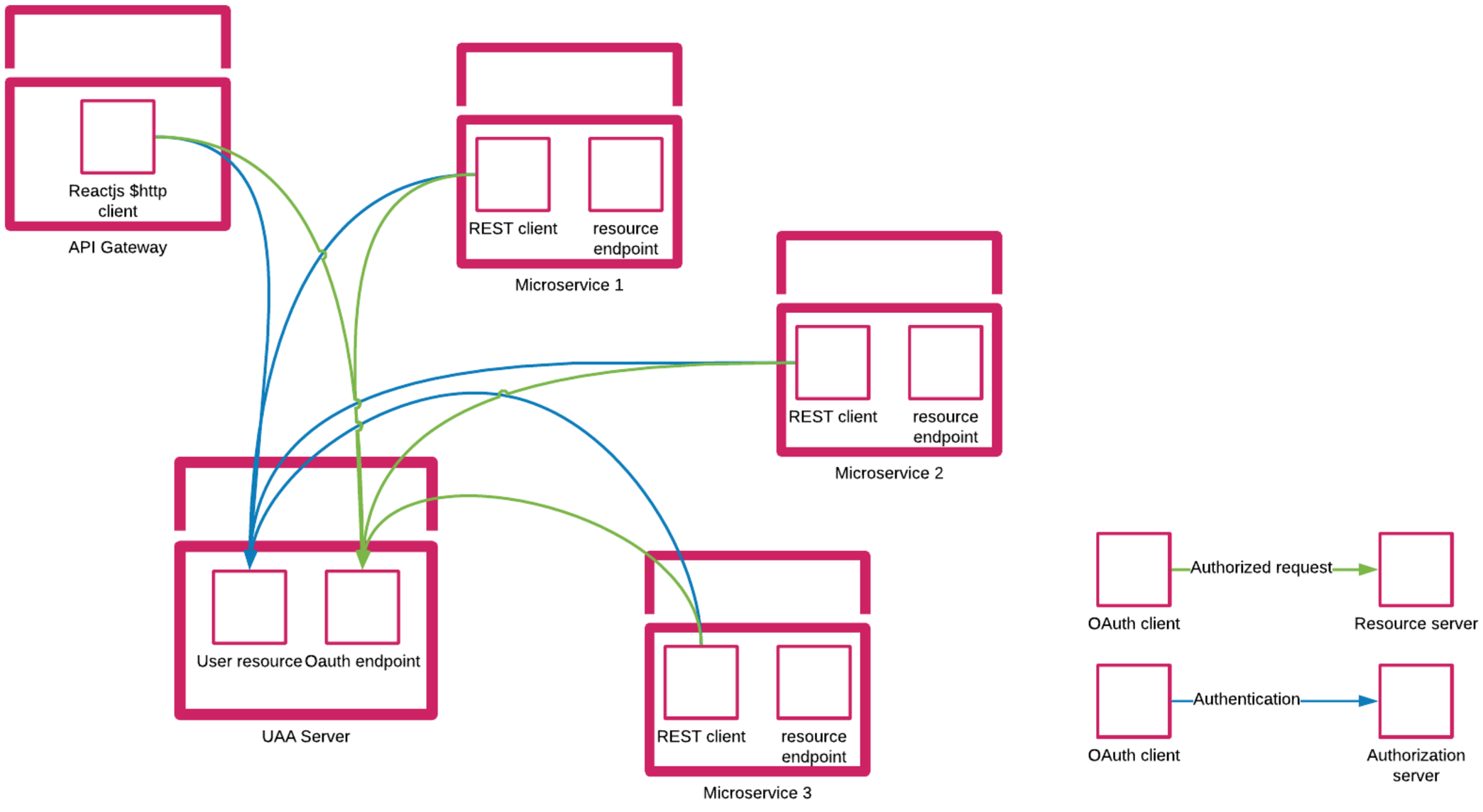
Демонстраційний плакат до магістерської дисертації

Структурна схема API Gateway сервісу

Виконав студент гр. ІП-82мп — Стеценко В.О.

Керівник — Ліщук К.І.

Структурна схема UAA серверу



Демонстраційний плакат до магістерської дисертації

Структурна схема UAA серверу

Виконав студент гр. ІП-82мп Стеценко В.О.

Керівник Ліщук К.І.

ДОДАТОК Б
ЛІСТИНГ ПРОГРАМИ

EventsSlavatoryGatewayApp

```
package org.events.salvatory;

import org.events.salvatory.config.ApplicationProperties;
import org.events.salvatory.config.DefaultProfileUtil;

import io.github.jhipster.config.JHipsterConstants;

import org.apache.commons.lang3.StringUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.liquibase.LiquibaseProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;
import org.springframework.core.env.Environment;

import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Arrays;
import java.util.Collection;

@SpringBootApplication
@EnableConfigurationProperties({LiquibaseProperties.class, ApplicationProperties.class})
@EnableDiscoveryClient
@EnableZuulProxy
public class EventsSlavatoryGatewayApp implements InitializingBean {

    private static final Logger log = LoggerFactory.getLogger(EventsSlavatoryGatewayApp.class);

    private final Environment env;

    public EventsSlavatoryGatewayApp(Environment env) {
        this.env = env;
    }

    /**
     * Initializes events_slavatory_gateway.
     * <p>
     * Spring profiles can be configured with a program argument --spring.profiles.active=your-active-profile
     * <p>
     * You can find more information on how profiles work with JHipster on <a
href="https://www.jhipster.tech/profiles/">https://www.jhipster.tech/profiles/</a>.
     */
    @Override
    public void afterPropertiesSet() throws Exception {
        Collection<String> activeProfiles = Arrays.asList(env.getActiveProfiles());
        if (activeProfiles.contains(JHipsterConstants.SPRING_PROFILE_DEVELOPMENT) &&
activeProfiles.contains(JHipsterConstants.SPRING_PROFILE_PRODUCTION)) {
            log.error("You have misconfigured your application! It should not run " +
                "with both the 'dev' and 'prod' profiles at the same time.");
        }
        if (activeProfiles.contains(JHipsterConstants.SPRING_PROFILE_DEVELOPMENT) &&
activeProfiles.contains(JHipsterConstants.SPRING_PROFILE_CLOUD)) {
            log.error("You have misconfigured your application! It should not " +
                "run with both the 'dev' and 'cloud' profiles at the same time.");
        }
    }

    /**
     * Main method, used to run the application.
     *
     * @param args the command line arguments.
     */
```

```
    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(EventsSlavatoryGatewayApp.class);
        DefaultProfileUtil.addDefaultProfile(app);
        Environment env = app.run(args).getEnvironment();
        logApplicationStartup(env);
    }

    private static void logApplicationStartup(Environment env) {
        String protocol = "http";
        if (env.getProperty("server.ssl.key-store") != null) {
            protocol = "https";
        }
        String serverPort = env.getProperty("server.port");
        String contextPath = env.getProperty("server.servlet.context-path");
        if (StringUtils.isBlank(contextPath)) {
            contextPath = "/";
        }
        String hostAddress = "localhost";
        try {
            hostAddress = InetAddress.getLocalHost().getHostAddress();
        } catch (UnknownHostException e) {
            log.warn("The host name could not be determined, using `localhost` as fallback");
        }
        log.info("\n----------------------------------------------------------\n\t" +
                "Application '{}' is running! Access URLs:\n\t" +
                "Local: \t\t{}://localhost:{}{}\n\t" +
                "External: \t{}://{}:{}{}\n\t" +
                "Profile(s): \t{}\n----------------------------------------------------------",
            env.getProperty("spring.application.name"),
            protocol,
            serverPort,
            contextPath,
            protocol,
            hostAddress,
            serverPort,
            contextPath,
            env.getActiveProfiles());

        String configServerStatus = env.getProperty("configserver.status");
        if (configServerStatus == null) {
            configServerStatus = "Not found or not setup for this application";
        }
        log.info("\n----------------------------------------------------------\n\t" +
                "Config Server: \t{}\n----------------------------------------------------------", configServerStatus);
    }
}
```

GatewayResource

```
package org.events.salvatory.web.rest;

import org.events.salvatory.web.rest.vm.RouteVM;

import java.util.ArrayList;
import java.util.List;

import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.cloud.netflix.zuul.filters.Route;
import org.springframework.cloud.netflix.zuul.filters.RouteLocator;
import org.springframework.http.*;
import org.springframework.security.access.annotation.Secured;
import org.events.salvatory.security.AuthoritiesConstants;
import org.springframework.web.bind.annotation.*;

/**
 * REST controller for managing Gateway configuration.
 */
@RestController
```

```
@RequestMapping("/api/gateway")
public class GatewayResource {

    private final RouteLocator routeLocator;

    private final DiscoveryClient discoveryClient;

    public GatewayResource(RouteLocator routeLocator, DiscoveryClient discoveryClient) {
        this.routeLocator = routeLocator;
        this.discoveryClient = discoveryClient;
    }

    /**
     * {@code GET  /routes} : get the active routes.
     *
     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and with body the list of routes.
     */
    @GetMapping("/routes")
    @Secured(AuthoritiesConstants.ADMIN)
    public ResponseEntity<List<RouteVM>> activeRoutes() {
        List<Route> routes = routeLocator.getRoutes();
        List<RouteVM> routeVMs = new ArrayList<>();
        routes.forEach(route -> {
            RouteVM routeVM = new RouteVM();
            routeVM.setPath(route.getFullPath());
            routeVM.setServiceId(route.getId());
            routeVM.setServiceInstances(discoveryClient.getInstances(route.getLocation()));
            routeVMs.add(routeVM);
        });
        return ResponseEntity.ok(routeVMs);
    }
}
```

AuthResource

```
package org.events.salvatory.web.rest;

import org.events.salvatory.security.oauth2.OAuth2AuthenticationService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.oauth2.common.OAuth2AccessToken;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Map;

/**
 * Authentication endpoint for web client.
 * Used to authenticate a user using OAuth2 access tokens or log him out.
 *
 * @author markus.oellinger
 */
@RestController
@RequestMapping("/auth")
public class AuthResource {

    private final Logger log = LoggerFactory.getLogger(AuthResource.class);

    private OAuth2AuthenticationService authenticationService;

    public AuthResource(OAuth2AuthenticationService authenticationService) {
        this.authenticationService = authenticationService;
```

```
    }

    /**
     * Authenticates a user setting the access and refresh token cookies.
     *
     * @param request  the {@link HttpServletRequest} holding - among others - the headers passed from the client.
     * @param response the {@link HttpServletResponse} getting the cookies set upon successful authentication.
     * @param params   the login params (username, password, rememberMe).
     * @return the access token of the authenticated user. Will return an error code if it fails to authenticate the user.
     */
    @RequestMapping(value = "/login", method = RequestMethod.POST, consumes = MediaType
        .APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<OAuth2AccessToken> authenticate(HttpServletRequest request, HttpServletResponse response,
@RequestBody
        Map<String, String> params) {
        return authenticationService.authenticate(request, response, params);
    }

    /**
     * Logout current user deleting his cookies.
     *
     * @param request  the {@link HttpServletRequest} holding - among others - the headers passed from the client.
     * @param response the {@link HttpServletResponse} getting the cookies set upon successful authentication.
     * @return an empty response entity.
     */
    @RequestMapping(value = "/logout", method = RequestMethod.POST)
    public ResponseEntity<?> logout(HttpServletRequest request, HttpServletResponse response) {
        log.info("logging out user {}", SecurityContextHolder.getContext().getAuthentication().getName());
        authenticationService.logout(request, response);
        return ResponseEntity.noContent().build();
    }
}
```

RefreshTokenFilter

```
package org.events.salvatory.web.filter;

import org.events.salvatory.security.oauth2.OAuth2AuthenticationService;
import org.events.salvatory.security.oauth2.OAuth2CookieHelper;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.oauth2.common.OAuth2AccessToken;
import org.springframework.security.oauth2.common.exceptions.ClientAuthenticationException;
import org.springframework.security.oauth2.common.exceptions.InvalidTokenException;
import org.springframework.security.oauth2.common.exceptions.UnauthorizedClientException;
import org.springframework.security.oauth2.provider.token.TokenStore;
import org.springframework.web.client.HttpClientErrorException;
import org.springframework.web.filter.GenericFilterBean;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

/**
 * Filters incoming requests and refreshes the access token before it expires.
 */
public class RefreshTokenFilter extends GenericFilterBean {
    /**
     * Number of seconds before expiry to start refreshing access tokens so we don't run into race conditions when forwarding
     * requests downstream. Otherwise, access tokens may still be valid when we check here but may then be expired
     * when relayed to another microservice a wee bit later.
     */
    private static final int REFRESH_WINDOW_SECS = 30;
```

```
    private final Logger log = LoggerFactory.getLogger(RefreshTokenFilter.class);

    /**
     * The {@link OAuth2AuthenticationService} is doing the actual work. We are just a simple filter after all.
     */
    private final OAuth2AuthenticationService authenticationService;
    private final TokenStore tokenStore;

    public RefreshTokenFilter(OAuth2AuthenticationService authenticationService, TokenStore tokenStore) {
        this.authenticationService = authenticationService;
        this.tokenStore = tokenStore;
    }

    /**
     * Check access token cookie and refresh it, if it is either not present, expired or about to expire.
     */
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
        throws IOException, ServletException {
        HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;
        HttpServletResponse httpServletResponse = (HttpServletResponse) servletResponse;
        try {
            httpServletRequest = refreshTokensIfExpiring(httpServletRequest, httpServletResponse);
        } catch (ClientAuthenticationException ex) {
            log.warn("Security exception: could not refresh tokens", ex);
            httpServletRequest = authenticationService.stripTokens(httpServletRequest);
        }
        filterChain.doFilter(httpServletRequest, servletResponse);
    }

    /**
     * Refresh the access and refresh tokens if they are about to expire.
     *
     * @param httpServletRequest  the servlet request holding the current cookies. If no refresh cookie is present,
     *                            then we are out of luck.
     * @param httpServletResponse the servlet response that gets the new set-cookie headers, if they had to be
     *                            refreshed.
     * @return a new request to use downstream that contains the new cookies, if they had to be refreshed.
     * @throws InvalidTokenException if the tokens could not be refreshed.
     */
    public HttpServletRequest refreshTokensIfExpiring(HttpServletRequest httpServletRequest, HttpServletResponse
        httpServletResponse) {
        HttpServletRequest newHttpServletRequest = httpServletRequest;
        //get access token from cookie
        Cookie accessTokenCookie = OAuth2CookieHelper.getAccessTokenCookie(httpServletRequest);
        if (mustRefreshToken(accessTokenCookie)) {        //we either have no access token, or it is expired, or it is about to
expire
            //get the refresh token cookie and, if present, request new tokens
            Cookie refreshCookie = OAuth2CookieHelper.getRefreshTokenCookie(httpServletRequest);
            if (refreshCookie != null) {
                try {
                    newHttpServletRequest = authenticationService.refreshToken(httpServletRequest, httpServletResponse,
refreshCookie);
                } catch (HttpClientErrorException ex) {
                    throw new UnauthorizedClientException("could not refresh OAuth2 token", ex);
                }
            } else if (accessTokenCookie != null) {
                log.warn("access token found, but no refresh token, stripping them all");
                OAuth2AccessToken token = tokenStore.readAccessToken(accessTokenCookie.getValue());
                if (token.isExpired()) {
                    throw new InvalidTokenException("access token has expired, but there's no refresh token");
                }
            }
        }
        return newHttpServletRequest;
    }
```

```
    /**
     * Check if we must refresh the access token.
     * We must refresh it, if we either have no access token, or it is expired, or it is about to expire.
     *
     * @param accessTokenCookie the current access token.
     * @return true, if it must be refreshed; false, otherwise.
     */
    private boolean mustRefreshToken(Cookie accessTokenCookie) {
        if (accessTokenCookie == null) {
            return true;
        }
        OAuth2AccessToken token = tokenStore.readAccessToken(accessTokenCookie.getValue());
        //check if token is expired or about to expire
        if (token.isExpired() || token.getExpiresIn() < REFRESH_WINDOW_SECS) {
            return true;
        }
        return false;       //access token is still fine
    }
}
```

RefreshTokenFilterConfigurer

```
package org.events.salvatory.web.filter;

import org.events.salvatory.security.oauth2.OAuth2AuthenticationService;

import org.springframework.security.config.annotation.SecurityConfigurerAdapter;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.oauth2.provider.authentication.OAuth2AuthenticationProcessingFilter;
import org.springframework.security.oauth2.provider.token.TokenStore;
import org.springframework.security.web.DefaultSecurityFilterChain;

/**
 * Configures a {@link RefreshTokenFilter} to refresh access tokens if they are about to expire.
 *
 * @see RefreshTokenFilter
 */
public class RefreshTokenFilterConfigurer extends SecurityConfigurerAdapter<DefaultSecurityFilterChain, HttpSecurity> {
    /**
     * {@link RefreshTokenFilter} needs the {@link OAuth2AuthenticationService} to refresh cookies using the refresh token.
     */
    private OAuth2AuthenticationService authenticationService;
    private final TokenStore tokenStore;

    public RefreshTokenFilterConfigurer(OAuth2AuthenticationService authenticationService, TokenStore tokenStore) {
        this.authenticationService = authenticationService;
        this.tokenStore = tokenStore;
    }

    /**
     * Install {@link RefreshTokenFilter} as a servlet Filter.
     */
    @Override
    public void configure(HttpSecurity http) throws Exception {
        RefreshTokenFilter customFilter = new RefreshTokenFilter(authenticationService, tokenStore);
        http.addFilterBefore(customFilter, OAuth2AuthenticationProcessingFilter.class);
    }
}
```

UaaTokenEndpointClient

```
package org.events.salvatory.security.oauth2;

import org.events.salvatory.config.oauth2.OAuth2Properties;
import io.github.jhipster.config.JHipsterProperties;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.http.HttpHeaders;
import org.springframework.stereotype.Component;
import org.springframework.util.Base64Utils;
import org.springframework.util.MultiValueMap;
import org.springframework.web.client.RestTemplate;
```

```
import java.nio.charset.StandardCharsets;

/**
 * Client talking to UAA's token endpoint to do different OAuth2 grants.
 */
@Component
public class UaaTokenEndpointClient extends OAuth2TokenEndpointClientAdapter implements
OAuth2TokenEndpointClient {

    public UaaTokenEndpointClient(@Qualifier("loadBalancedRestTemplate") RestTemplate restTemplate,
                                  JHipsterProperties jHipsterProperties, OAuth2Properties oAuth2Properties) {
        super(restTemplate, jHipsterProperties, oAuth2Properties);
    }

    @Override
    protected void addAuthentication(HttpHeaders reqHeaders, MultiValueMap<String, String> formParams) {
        reqHeaders.add("Authorization", getAuthorizationHeader());
    }

    /**
     * @return a Basic authorization header to be used to talk to UAA.
     */
    protected String getAuthorizationHeader() {
        String clientId = getClientId();
        String clientSecret = getClientSecret();
        String authorization = clientId + ":" + clientSecret;
        return "Basic " + Base64Utils.encodeToString(authorization.getBytes(StandardCharsets.UTF_8));
    }

}
UaaSignatureVerifierClient
package org.events.salvatory.security.oauth2;

import org.events.salvatory.config.oauth2.OAuth2Properties;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.security.jwt.crypto.sign.RsaVerifier;
import org.springframework.security.jwt.crypto.sign.SignatureVerifier;
import org.springframework.security.oauth2.common.exceptions.InvalidClientException;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import java.util.Map;

/**
 * Client fetching the public key from UAA to create a {@link SignatureVerifier}.
 */
@Component
public class UaaSignatureVerifierClient implements OAuth2SignatureVerifierClient {
    private final Logger log = LoggerFactory.getLogger(UaaSignatureVerifierClient.class);
    private final RestTemplate restTemplate;
    protected final OAuth2Properties oAuth2Properties;

    public UaaSignatureVerifierClient(DiscoveryClient discoveryClient, @Qualifier("loadBalancedRestTemplate")
RestTemplate restTemplate,
                                      OAuth2Properties oAuth2Properties) {
        this.restTemplate = restTemplate;
        this.oAuth2Properties = oAuth2Properties;
        // Load available UAA servers
        discoveryClient.getServices();
    }
```

```
    /**
     * Fetches the public key from the UAA.
     *
     * @return the public key used to verify JWT tokens; or {@code null}.
     */
    @Override
    public SignatureVerifier getSignatureVerifier() throws Exception {
        try {
            HttpEntity<Void> request = new HttpEntity<Void>(new HttpHeaders());
            String key = (String) restTemplate
                .exchange(getPublicKeyEndpoint(), HttpMethod.GET, request, Map.class).getBody()
                .get("value");
            return new RsaVerifier(key);
        } catch (IllegalStateException ex) {
            log.warn("could not contact UAA to get public key");
            return null;
        }
    }

    /**
     * Returns the configured endpoint URI to retrieve the public key.
     *
     * @return the configured endpoint URI to retrieve the public key.
     */
    private String getPublicKeyEndpoint() {
        String tokenEndpointUrl = oAuth2Properties.getSignatureVerification().getPublicKeyEndpointUri();
        if (tokenEndpointUrl == null) {
            throw new InvalidClientException("no token endpoint configured in application properties");
        }
        return tokenEndpointUrl;
    }
}
```

OAuth2TokenEndpointClientAdapter

```
package org.events.salvatory.security.oauth2;

import org.events.salvatory.config.oauth2.OAuth2Properties;
import io.github.jhipster.config.JHipsterProperties;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.security.oauth2.common.OAuth2AccessToken;
import org.springframework.security.oauth2.common.exceptions.InvalidClientException;
import org.springframework.util.LinkedMultiValueMap;
import org.springframework.util.MultiValueMap;
import org.springframework.web.client.HttpClientErrorException;
import org.springframework.web.client.RestTemplate;

/**
 * Default base class for an {@link OAuth2TokenEndpointClient}.
 * Individual implementations for a particular OAuth2 provider can use this as a starting point.
 */
public abstract class OAuth2TokenEndpointClientAdapter implements OAuth2TokenEndpointClient {
    private final Logger log = LoggerFactory.getLogger(OAuth2TokenEndpointClientAdapter.class);
    protected final RestTemplate restTemplate;
    protected final JHipsterProperties jHipsterProperties;
    protected final OAuth2Properties oAuth2Properties;

    public OAuth2TokenEndpointClientAdapter(RestTemplate restTemplate, JHipsterProperties jHipsterProperties,
OAuth2Properties oAuth2Properties) {
        this.restTemplate = restTemplate;
        this.jHipsterProperties = jHipsterProperties;
        this.oAuth2Properties = oAuth2Properties;
```

```
    }

    /**
     * Sends a password grant to the token endpoint.
     *
     * @param username the username to authenticate.
     * @param password his password.
     * @return the access token.
     */
    @Override
    public OAuth2AccessToken sendPasswordGrant(String username, String password) {
        HttpHeaders reqHeaders = new HttpHeaders();
        reqHeaders.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
        MultiValueMap<String, String> formParams = new LinkedMultiValueMap<>();
        formParams.set("username", username);
        formParams.set("password", password);
        formParams.set("grant_type", "password");
        addAuthentication(reqHeaders, formParams);
        HttpEntity<MultiValueMap<String, String>> entity = new HttpEntity<>(formParams, reqHeaders);
        log.debug("contacting OAuth2 token endpoint to login user: {}", username);
        ResponseEntity<OAuth2AccessToken>
            responseEntity = restTemplate.postForEntity(getTokenEndpoint(), entity, OAuth2AccessToken.class);
        if (responseEntity.getStatusCode() != HttpStatus.OK) {
            log.debug("failed to authenticate user with OAuth2 token endpoint, status: {}",
responseEntity.getStatusCodeValue());
            throw new HttpClientErrorException(responseEntity.getStatusCode());
        }
        OAuth2AccessToken accessToken = responseEntity.getBody();
        return accessToken;
    }

    /**
     * Sends a refresh grant to the token endpoint using the current refresh token to obtain new tokens.
     *
     * @param refreshTokenValue the refresh token to use to obtain new tokens.
     * @return the new, refreshed access token.
     */
    @Override
    public OAuth2AccessToken sendRefreshGrant(String refreshTokenValue) {
        MultiValueMap<String, String> params = new LinkedMultiValueMap<>();
        params.add("grant_type", "refresh_token");
        params.add("refresh_token", refreshTokenValue);
        HttpHeaders headers = new HttpHeaders();
        addAuthentication(headers, params);
        HttpEntity<MultiValueMap<String, String>> entity = new HttpEntity<>(params, headers);
        log.debug("contacting OAuth2 token endpoint to refresh OAuth2 JWT tokens");
        ResponseEntity<OAuth2AccessToken> responseEntity = restTemplate.postForEntity(getTokenEndpoint(), entity,
                                                            OAuth2AccessToken.class);
        if (responseEntity.getStatusCode() != HttpStatus.OK) {
            log.debug("failed to refresh tokens: {}", responseEntity.getStatusCodeValue());
            throw new HttpClientErrorException(responseEntity.getStatusCode());
        }
        OAuth2AccessToken accessToken = responseEntity.getBody();
        log.info("refreshed OAuth2 JWT cookies using refresh_token grant");
        return accessToken;
    }

    protected abstract void addAuthentication(HttpHeaders reqHeaders, MultiValueMap<String, String> formParams);

    protected String getClientSecret() {
        String clientSecret = oAuth2Properties.getWebClientConfiguration().getSecret();
        if (clientSecret == null) {
            throw new InvalidClientException("no client-secret configured in application properties");
        }
        return clientSecret;
    }
```

```
    protected String getClientId() {
        String clientId = oAuth2Properties.getWebClientConfiguration().getClientId();
        if (clientId == null) {
            throw new InvalidClientException("no client-id configured in application properties");
        }
        return clientId;
    }

    /**
     * Returns the configured OAuth2 token endpoint URI.
     *
     * @return the OAuth2 token endpoint URI.
     */
    protected String getTokenEndpoint() {
        String tokenEndpointUrl = jHipsterProperties.getSecurity().getClientAuthorization().getAccessTokenUri();
        if (tokenEndpointUrl == null) {
            throw new InvalidClientException("no token endpoint configured in application properties");
        }
        return tokenEndpointUrl;
    }

}
OAuth2TokenEndpointClient
package org.events.salvatory.security.oauth2;

import org.springframework.security.oauth2.common.OAuth2AccessToken;

/**
 * Client talking to an OAuth2 Authorization server token endpoint.
 *
 * @see UaaTokenEndpointClient
 * @see OAuth2TokenEndpointClientAdapter
 */
public interface OAuth2TokenEndpointClient {
    /**
     * Send a password grant to the token endpoint.
     *
     * @param username the username to authenticate.
     * @param password his password.
     * @return the access token and enclosed refresh token received from the token endpoint.
     * @throws org.springframework.security.oauth2.common.exceptions.ClientAuthenticationException
     * if we cannot contact the token endpoint.
     */
    OAuth2AccessToken sendPasswordGrant(String username, String password);

    /**
     * Send a refresh_token grant to the token endpoint.
     *
     * @param refreshTokenValue the refresh token used to get new tokens.
     * @return the new access/refresh token pair.
     * @throws org.springframework.security.oauth2.common.exceptions.ClientAuthenticationException
     * if we cannot contact the token endpoint.
     */
    OAuth2AccessToken sendRefreshGrant(String refreshTokenValue);
}
OAuth2Cookies
package org.events.salvatory.security.oauth2;

import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletResponse;

/**
 * Holds the access token and refresh token cookies.
 */
class OAuth2Cookies {
    private Cookie accessTokenCookie;
    private Cookie refreshTokenCookie;
```

```
    public Cookie getAccessTokenCookie() {
        return accessTokenCookie;
    }

    public Cookie getRefreshTokenCookie() {
        return refreshTokenCookie;
    }

    public void setCookies(Cookie accessTokenCookie, Cookie refreshTokenCookie) {
        this.accessTokenCookie = accessTokenCookie;
        this.refreshTokenCookie = refreshTokenCookie;
    }

    /**
     * Add the access token and refresh token as cookies to the response after successful authentication.
     *
     * @param response the response to add them to.
     */
    void addCookiesTo(HttpServletResponse response) {
        response.addCookie(getAccessTokenCookie());
        response.addCookie(getRefreshTokenCookie());
    }
}
```

OAuth2AuthenticationService

```
package org.events.salvatory.security.oauth2;

import io.github.jhipster.security.PersistentTokenCache;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.oauth2.common.OAuth2AccessToken;
import org.springframework.web.client.HttpClientErrorException;

import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Map;

/**
 * Manages authentication cases for OAuth2 updating the cookies holding access and refresh tokens accordingly.
 * <p>
 * It can authenticate users, refresh the token cookies should they expire and log users out.
 */
public class OAuth2AuthenticationService {

    private final Logger log = LoggerFactory.getLogger(OAuth2AuthenticationService.class);

    /**
     * Number of milliseconds to cache refresh token grants so we don't have to repeat them in case of parallel requests.
     */
    private static final long REFRESH_TOKEN_VALIDITY_MILLIS = 10000l;

    /**
     * Used to contact the OAuth2 token endpoint.
     */
    private final OAuth2TokenEndpointClient authorizationClient;

    /**
     * Helps us with cookie handling.
     */
    private final OAuth2CookieHelper cookieHelper;

    /**
     * Caches Refresh grant results for a refresh token value so we can reuse them.
     * This avoids hammering UAA in case of several multi-threaded requests arriving in parallel.
```

```
    */
   private final PersistentTokenCache<OAuth2Cookies> recentlyRefreshed;

   public OAuth2AuthenticationService(OAuth2TokenEndpointClient authorizationClient, OAuth2CookieHelper
cookieHelper) {
      this.authorizationClient = authorizationClient;
      this.cookieHelper = cookieHelper;
      recentlyRefreshed = new PersistentTokenCache<>(REFRESH_TOKEN_VALIDITY_MILLIS);
   }

   /**
    * Authenticate the user by username and password.
    *
    * @param request  the request coming from the client.
    * @param response the response going back to the server.
    * @param params   the params holding the username, password and rememberMe.
    * @return the {@link OAuth2AccessToken} as a {@link ResponseEntity}. Will return {@code OK (200)}, if successful.
    * If the UAA cannot authenticate the user, the status code returned by UAA will be returned.
    */
   public ResponseEntity<OAuth2AccessToken> authenticate(HttpServletRequest request, HttpServletResponse response,
                                         Map<String, String> params) {
      try {
         String username = params.get("username");
         String password = params.get("password");
         boolean rememberMe = Boolean.valueOf(params.get("rememberMe"));
         OAuth2AccessToken accessToken = authorizationClient.sendPasswordGrant(username, password);
         OAuth2Cookies cookies = new OAuth2Cookies();
         cookieHelper.createCookies(request, accessToken, rememberMe, cookies);
         cookies.addCookiesTo(response);
         if (log.isDebugEnabled()) {
            log.debug("successfully authenticated user {}", params.get("username"));
         }
         return ResponseEntity.ok(accessToken);
      } catch (HttpClientErrorException ex) {
         log.error("failed to get OAuth2 tokens from UAA", ex);
         throw new BadCredentialsException("Invalid credentials");
      }
   }

   /**
    * Try to refresh the access token using the refresh token provided as cookie.
    * Note that browsers typically send multiple requests in parallel which means the access token
    * will be expired on multiple threads. We don't want to send multiple requests to UAA though,
    * so we need to cache results for a certain duration and synchronize threads to avoid sending
    * multiple requests in parallel.
    *
    * @param request       the request potentially holding the refresh token.
    * @param response      the response setting the new cookies (if refresh was successful).
    * @param refreshCookie the refresh token cookie. Must not be null.
    * @return the new servlet request containing the updated cookies for relaying downstream.
    */
   public HttpServletRequest refreshToken(HttpServletRequest request, HttpServletResponse response, Cookie
      refreshCookie) {
      //check if non-remember-me session has expired
      if (cookieHelper.isSessionExpired(refreshCookie)) {
         log.info("session has expired due to inactivity");
         logout(request, response);       //logout to clear cookies in browser
         return stripTokens(request);          //don't include cookies downstream
      }
      OAuth2Cookies cookies = getCachedCookies(refreshCookie.getValue());
      synchronized (cookies) {
         //check if we have a result from another thread already
         if (cookies.getAccessTokenCookie() == null) {          //no, we are first!
            //send a refresh_token grant to UAA, getting new tokens
            String refreshCookieValue = OAuth2CookieHelper.getRefreshTokenValue(refreshCookie);
            OAuth2AccessToken accessToken = authorizationClient.sendRefreshGrant(refreshCookieValue);
            boolean rememberMe = OAuth2CookieHelper.isRememberMe(refreshCookie);
```

```
            cookieHelper.createCookies(request, accessToken, rememberMe, cookies);
            //add cookies to response to update browser
            cookies.addCookiesTo(response);
         } else {
            log.debug("reusing cached refresh_token grant");
         }
         //replace cookies in original request with new ones
         CookieCollection requestCookies = new CookieCollection(request.getCookies());
         requestCookies.add(cookies.getAccessTokenCookie());
         requestCookies.add(cookies.getRefreshTokenCookie());
         return new CookiesHttpServletRequestWrapper(request, requestCookies.toArray());
      }
   }

   /**
    * Get the result from the cache in a thread-safe manner.
    *
    * @param refreshTokenValue the refresh token for which we want the results.
    * @return a RefreshGrantResult for that token. This will either be empty, if we are the first one to do the
    * request, or contain some results already, if another thread already handled the grant for us.
    */
   private OAuth2Cookies getCachedCookies(String refreshTokenValue) {
      synchronized (recentlyRefreshed) {
         OAuth2Cookies ctx = recentlyRefreshed.get(refreshTokenValue);
         if (ctx == null) {
            ctx = new OAuth2Cookies();
            recentlyRefreshed.put(refreshTokenValue, ctx);
         }
         return ctx;
      }
   }

   /**
    * Logs the user out by clearing all cookies.
    *
    * @param httpServletRequest  the request containing the Cookies.
    * @param httpServletResponse the response used to clear them.
    */
   public void logout(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse) {
      cookieHelper.clearCookies(httpServletRequest, httpServletResponse);
   }

   /**
    * Strips token cookies preventing them from being used further down the chain.
    * For example, the OAuth2 client won't checked them and they won't be relayed to other services.
    *
    * @param httpServletRequest the incoming request.
    * @return the request to replace it with which has the tokens stripped.
    */
   public HttpServletRequest stripTokens(HttpServletRequest httpServletRequest) {
      Cookie[] cookies = cookieHelper.stripCookies(httpServletRequest.getCookies());
      return new CookiesHttpServletRequestWrapper(httpServletRequest, cookies);
   }
}
```

CookieCollection

```
package org.events.salvatory.security.oauth2;

import javax.servlet.http.Cookie;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

/**
 * A {@link Collection} of {@link Cookie}s that allows modification - unlike a mere array.
 * <p>
```

```
 * Since {@link Cookie} doesn't implement {@code hashCode} nor {@code equals},
 * we cannot simply put it into a {@code HashSet}.
 */
public class CookieCollection implements Collection<Cookie> {
    private final Map<String, Cookie> cookieMap;

    public CookieCollection() {
        cookieMap = new HashMap<>();
    }

    public CookieCollection(Cookie... cookies) {
        this(Arrays.asList(cookies));
    }

    public CookieCollection(Collection<? extends Cookie> cookies) {
        cookieMap = new HashMap<>(cookies.size());
        addAll(cookies);
    }

    @Override
    public int size() {
        return cookieMap.size();
    }

    @Override
    public boolean isEmpty() {
        return cookieMap.isEmpty();
    }

    @Override
    public boolean contains(Object o) {
        if (o instanceof String) {
            return cookieMap.containsKey(o);
        }
        if (o instanceof Cookie) {
            return cookieMap.containsValue(o);
        }
        return false;
    }

    @Override
    public Iterator<Cookie> iterator() {
        return cookieMap.values().iterator();
    }

    public Cookie[] toArray() {
        Cookie[] cookies = new Cookie[cookieMap.size()];
        return toArray(cookies);
    }

    @Override
    public <T> T[] toArray(T[] ts) {
        return cookieMap.values().toArray(ts);
    }

    @Override
    public boolean add(Cookie cookie) {
        if (cookie == null) {
            return false;
        }
        cookieMap.put(cookie.getName(), cookie);
        return true;
    }

    @Override
    public boolean remove(Object o) {
        if (o instanceof String) {
```

```
            return cookieMap.remove(o) != null;
        }
        if (o instanceof Cookie) {
            Cookie c = (Cookie)o;
            return cookieMap.remove(c.getName()) != null;
        }
        return false;
    }

    public Cookie get(String name) {
        return cookieMap.get(name);
    }

    @Override
    public boolean containsAll(Collection<?> collection) {
        for(Object o : collection) {
            if (!contains(o)) {
                return false;
            }
        }
        return true;
    }

    @Override
    public boolean addAll(Collection<? extends Cookie> collection) {
        boolean result = false;
        for(Cookie cookie : collection) {
            result|= add(cookie);
        }
        return result;
    }

    @Override
    public boolean removeAll(Collection<?> collection) {
        boolean result = false;
        for(Object cookie : collection) {
            result|= remove(cookie);
        }
        return result;
    }

    @Override
    public boolean retainAll(Collection<?> collection) {
        boolean result = false;
        Iterator<Map.Entry<String, Cookie>> it = cookieMap.entrySet().iterator();
        while(it.hasNext()) {
            Map.Entry<String, Cookie> e = it.next();
            if (!collection.contains(e.getKey()) && !collection.contains(e.getValue())) {
                it.remove();
                result = true;
            }
        }
        return result;
    }

    @Override
    public void clear() {
        cookieMap.clear();
    }
}
```

AccessControlFilter

```
package org.events.salvatory.gateway.accesscontrol;

import io.github.jhipster.config.JHipsterProperties;

import java.util.List;
import java.util.Map;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cloud.netflix.zuul.filters.Route;
import org.springframework.cloud.netflix.zuul.filters.RouteLocator;
import org.springframework.http.HttpStatus;

import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;

/**
 * Zuul filter for restricting access to backend micro-services endpoints.
 */
public class AccessControlFilter extends ZuulFilter {

    private final Logger log = LoggerFactory.getLogger(AccessControlFilter.class);

    private final RouteLocator routeLocator;

    private final JHipsterProperties jHipsterProperties;

    public AccessControlFilter(RouteLocator routeLocator, JHipsterProperties jHipsterProperties) {
        this.routeLocator = routeLocator;
        this.jHipsterProperties = jHipsterProperties;
    }

    @Override
    public String filterType() {
        return "pre";
    }

    @Override
    public int filterOrder() {
        return 0;
    }

    /**
     * Filter requests on endpoints that are not in the list of authorized microservices endpoints.
     */
    @Override
    public boolean shouldFilter() {
        String requestUri = RequestContext.getCurrentContext().getRequest().getRequestURI();
        String contextPath = RequestContext.getCurrentContext().getRequest().getContextPath();

        // If the request Uri does not start with the path of the authorized endpoints, we block the request
        for (Route route : routeLocator.getRoutes()) {
            String serviceUrl = contextPath + route.getFullPath();
            String serviceName = route.getId();

            // If this route correspond to the current request URI
            // We do a substring to remove the "**" at the end of the route URL
            if (requestUri.startsWith(serviceUrl.substring(0, serviceUrl.length() - 2))) {
            return !isAuthorizedRequest(serviceUrl, serviceName, requestUri);
            }
        }
        return true;
    }

    private boolean isAuthorizedRequest(String serviceUrl, String serviceName, String requestUri) {
        Map<String, List<String>> authorizedMicroservicesEndpoints = jHipsterProperties.getGateway()
            .getAuthorizedMicroservicesEndpoints();

        // If the authorized endpoints list was left empty for this route, all access are allowed
        if (authorizedMicroservicesEndpoints.get(serviceName) == null) {
            log.debug("Access Control: allowing access for {}, as no access control policy has been set up for " +
                "service: {}", requestUri, serviceName);
            return true;
```

```
        } else {
            List<String> authorizedEndpoints = authorizedMicroservicesEndpoints.get(serviceName);

            // Go over the authorized endpoints to control that the request URI matches it
            for (String endpoint : authorizedEndpoints) {
                // We do a substring to remove the "**/" at the end of the route URL
                String gatewayEndpoint = serviceUrl.substring(0, serviceUrl.length() - 3) + endpoint;
                if (requestUri.startsWith(gatewayEndpoint)) {
                    log.debug("Access Control: allowing access for {}, as it matches the following authorized " +
                        "microservice endpoint: {}", requestUri, gatewayEndpoint);
                    return true;
                }
            }
        }
        return false;
    }

    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        ctx.setResponseStatusCode(HttpStatus.FORBIDDEN.value());
        ctx.setSendZuulResponse(false);
        log.debug("Access Control: filtered unauthorized access on endpoint {}", ctx.getRequest().getRequestURI());
        return null;
    }
}
```

WebConfigurer

```
package org.events.salvatory.config;

import io.github.jhipster.config.JHipsterConstants;
import io.github.jhipster.config.JHipsterProperties;
import io.github.jhipster.config.h2.H2ConfigurationHelper;
import io.github.jhipster.web.filter.CachingHttpHeadersFilter;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.web.server.*;
import org.springframework.boot.web.servlet.ServletContextInitializer;
import org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.Environment;
import org.springframework.core.env.Profiles;
import org.springframework.http.MediaType;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import org.springframework.web.filter.CorsFilter;

import javax.servlet.*;
import java.io.File;
import java.io.UnsupportedEncodingException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Paths;
import java.util.*;

import static java.net.URLDecoder.decode;

/**
 * Configuration of web application with Servlet 3.0 APIs.
 */
@Configuration
public class WebConfigurer implements ServletContextInitializer, WebServerFactoryCustomizer<WebServerFactory> {

    private final Logger log = LoggerFactory.getLogger(WebConfigurer.class);

    private final Environment env;

    private final JHipsterProperties jHipsterProperties;
```

```
    public WebConfigurer(Environment env, JHipsterProperties jHipsterProperties) {
        this.env = env;
        this.jHipsterProperties = jHipsterProperties;
    }

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        if (env.getActiveProfiles().length != 0) {
            log.info("Web application configuration, using profiles: {}", (Object[]) env.getActiveProfiles());
        }
        EnumSet<DispatcherType> disps = EnumSet.of(DispatcherType.REQUEST, DispatcherType.FORWARD,
DispatcherType.ASYNC);
        if (env.acceptsProfiles(Profiles.of(JHipsterConstants.SPRING_PROFILE_PRODUCTION))) {
            initCachingHttpHeadersFilter(servletContext, disps);
        }
        if (env.acceptsProfiles(Profiles.of(JHipsterConstants.SPRING_PROFILE_DEVELOPMENT))) {
            initH2Console(servletContext);
        }
        log.info("Web application fully configured");
    }

    /**
     * Customize the Servlet engine: Mime types, the document root, the cache.
     */
    @Override
    public void customize(WebServerFactory server) {
        setMimeMappings(server);
        // When running in an IDE or with ./mvnw spring-boot:run, set location of the static web assets.
        setLocationForStaticAssets(server);
    }

    private void setMimeMappings(WebServerFactory server) {
        if (server instanceof ConfigurableServletWebServerFactory) {
            MimeMappings mappings = new MimeMappings(MimeMappings.DEFAULT);
            // IE issue, see https://github.com/jhipster/generator-jhipster/pull/711
            mappings.add("html", MediaType.TEXT_HTML_VALUE + ";charset=" +
StandardCharsets.UTF_8.name().toLowerCase());
            // CloudFoundry issue, see https://github.com/cloudfoundry/gorouter/issues/64
            mappings.add("json", MediaType.TEXT_HTML_VALUE + ";charset=" +
StandardCharsets.UTF_8.name().toLowerCase());
            ConfigurableServletWebServerFactory servletWebServer = (ConfigurableServletWebServerFactory) server;
            servletWebServer.setMimeMappings(mappings);
        }
    }

    private void setLocationForStaticAssets(WebServerFactory server) {
        if (server instanceof ConfigurableServletWebServerFactory) {
            ConfigurableServletWebServerFactory servletWebServer = (ConfigurableServletWebServerFactory) server;
            File root;
            String prefixPath = resolvePathPrefix();
            root = new File(prefixPath + "target/classes/static/");
            if (root.exists() && root.isDirectory()) {
                servletWebServer.setDocumentRoot(root);
            }
        }
    }

    /**
     * Resolve path prefix to static resources.
     */
    private String resolvePathPrefix() {
        String fullExecutablePath;
        try {
            fullExecutablePath = decode(this.getClass().getResource("").getPath(), StandardCharsets.UTF_8.name());
        } catch (UnsupportedEncodingException e) {
            /* try without decoding if this ever happens */
```

```
            fullExecutablePath = this.getClass().getResource("").getPath();
        }
        String rootPath = Paths.get(".").toUri().normalize().getPath();
        String extractedPath = fullExecutablePath.replace(rootPath, "");
        int extractionEndIndex = extractedPath.indexOf("target/");
        if (extractionEndIndex <= 0) {
            return "";
        }
        return extractedPath.substring(0, extractionEndIndex);
    }

    /**
     * Initializes the caching HTTP Headers Filter.
     */
    private void initCachingHttpHeadersFilter(ServletContext servletContext,
                                              EnumSet<DispatcherType> disps) {
        log.debug("Registering Caching HTTP Headers Filter");
        FilterRegistration.Dynamic cachingHttpHeadersFilter =
            servletContext.addFilter("cachingHttpHeadersFilter",
                new CachingHttpHeadersFilter(jHipsterProperties));

        cachingHttpHeadersFilter.addMappingForUrlPatterns(disps, true, "/i18n/*");
        cachingHttpHeadersFilter.addMappingForUrlPatterns(disps, true, "/content/*");
        cachingHttpHeadersFilter.addMappingForUrlPatterns(disps, true, "/app/*");
        cachingHttpHeadersFilter.setAsyncSupported(true);
    }

    @Bean
    public CorsFilter corsFilter() {
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        CorsConfiguration config = jHipsterProperties.getCors();
        if (config.getAllowedOrigins() != null && !config.getAllowedOrigins().isEmpty()) {
            log.debug("Registering CORS filter");
            source.registerCorsConfiguration("/api/**", config);
            source.registerCorsConfiguration("/management/**", config);
            source.registerCorsConfiguration("/v2/api-docs", config);
            source.registerCorsConfiguration("/auth/**", config);
            source.registerCorsConfiguration("/*/api/**", config);
            source.registerCorsConfiguration("/services/*/api/**", config);
            source.registerCorsConfiguration("/*/management/**", config);
            source.registerCorsConfiguration("/*/oauth/**", config);
        }
        return new CorsFilter(source);
    }

    /**
     * Initializes H2 console.
     */
    private void initH2Console(ServletContext servletContext) {
        log.debug("Initialize H2 console");
        H2ConfigurationHelper.initH2Console(servletContext);
    }

}
```

SecurityConfiguration

```
package org.events.salvatory.config;

import org.events.salvatory.config.oauth2.OAuth2JwtAccessTokenConverter;
import org.events.salvatory.config.oauth2.OAuth2Properties;
import org.events.salvatory.security.oauth2.OAuth2SignatureVerifierClient;
import org.events.salvatory.security.AuthoritiesConstants;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.cloud.client.loadbalancer.RestTemplateCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;
```

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer;
import org.springframework.security.oauth2.config.annotation.web.configuration.ResourceServerConfigurerAdapter;
import org.springframework.security.oauth2.provider.token.TokenStore;
import org.springframework.security.oauth2.provider.token.store.JwtAccessTokenConverter;
import org.springframework.security.oauth2.provider.token.store.JwtTokenStore;
import org.springframework.security.web.csrf.CookieCsrfTokenRepository;
import org.springframework.security.web.csrf.CsrfFilter;
import org.springframework.web.filter.CorsFilter;
import org.springframework.web.client.RestTemplate;

@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)
public class SecurityConfiguration extends ResourceServerConfigurerAdapter {
    private final OAuth2Properties oAuth2Properties;

    private final CorsFilter corsFilter;

    public SecurityConfiguration(OAuth2Properties oAuth2Properties, CorsFilter corsFilter) {
        this.oAuth2Properties = oAuth2Properties;
        this.corsFilter = corsFilter;
    }

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
            .csrf()
            .ignoringAntMatchers("/h2-console/**")
            .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
        .and()
            .addFilterBefore(corsFilter, CsrfFilter.class)
            .headers()
            .frameOptions()
            .disable()
        .and()
            .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and()
            .authorizeRequests()
            .antMatchers("/api/**").authenticated()
            .antMatchers("/management/health").permitAll()
            .antMatchers("/management/info").permitAll()
            .antMatchers("/management/prometheus").permitAll()
            .antMatchers("/management/**").hasAuthority(AuthoritiesConstants.ADMIN);
    }

    @Bean
    public TokenStore tokenStore(JwtAccessTokenConverter jwtAccessTokenConverter) {
        return new JwtTokenStore(jwtAccessTokenConverter);
    }

    @Bean
    public JwtAccessTokenConverter jwtAccessTokenConverter(OAuth2SignatureVerifierClient signatureVerifierClient) {
        return new OAuth2JwtAccessTokenConverter(oAuth2Properties, signatureVerifierClient);
    }

    @Bean
    @Qualifier("loadBalancedRestTemplate")
    public RestTemplate loadBalancedRestTemplate(RestTemplateCustomizer customizer) {
        RestTemplate restTemplate = new RestTemplate();
        customizer.customize(restTemplate);
        return restTemplate;
    }

    @Bean
```

```
    @Qualifier("vanillaRestTemplate")
    public RestTemplate vanillaRestTemplate() {
        return new RestTemplate();
    }
}
```

LoggingConfiguration

```
package org.events.salvatory.config;

import ch.qos.logback.classic.LoggerContext;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import io.github.jhipster.config.JHipsterProperties;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.ObjectProvider;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.info.BuildProperties;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.context.annotation.Configuration;

import java.util.HashMap;
import java.util.Map;

import static io.github.jhipster.config.logging.LoggingUtils.*;

/*
 * Configures the console and Logstash log appenders from the app properties
 */
@Configuration
@RefreshScope
public class LoggingConfiguration {

    public LoggingConfiguration(@Value("${spring.application.name}") String appName,
                                @Value("${server.port}") String serverPort,
                                JHipsterProperties jHipsterProperties,
                                ObjectProvider<BuildProperties> buildProperties,
                                ObjectMapper mapper) throws JsonProcessingException {

        LoggerContext context = (LoggerContext) LoggerFactory.getILoggerFactory();

        Map<String, String> map = new HashMap<>();
        map.put("app_name", appName);
        map.put("app_port", serverPort);
        buildProperties.ifAvailable(it -> map.put("version", it.getVersion()));
        String customFields = mapper.writeValueAsString(map);

        JHipsterProperties.Logging loggingProperties = jHipsterProperties.getLogging();
        JHipsterProperties.Logging.Logstash logstashProperties = loggingProperties.getLogstash();

        if (loggingProperties.isUseJsonFormat()) {
            addJsonConsoleAppender(context, customFields);
        }
        if (logstashProperties.isEnabled()) {
            addLogstashTcpSocketAppender(context, customFields, logstashProperties);
        }
        if (loggingProperties.isUseJsonFormat() || logstashProperties.isEnabled()) {
            addContextListener(context, customFields, loggingProperties);
        }
        if (jHipsterProperties.getMetrics().getLogs().isEnabled()) {
            setMetricsMarkerLogbackFilter(context, loggingProperties.isUseJsonFormat());
        }
    }
}
```

GatewayConfiguration

```
package org.events.salvatory.config;

import io.github.jhipster.config.JHipsterProperties;
```

```
import org.events.salvatory.gateway.accesscontrol.AccessControlFilter;
import org.events.salvatory.gateway.responserewriting.SwaggerBasePathRewritingFilter;
import org.springframework.cloud.netflix.zuul.filters.RouteLocator;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class GatewayConfiguration {

    @Configuration
    public static class SwaggerBasePathRewritingConfiguration {

        @Bean
        public SwaggerBasePathRewritingFilter swaggerBasePathRewritingFilter(){
            return new SwaggerBasePathRewritingFilter();
        }
    }

    @Configuration
    public static class AccessControlFilterConfiguration {

        @Bean
        public AccessControlFilter accessControlFilter(RouteLocator routeLocator, JHipsterProperties jHipsterProperties){
            return new AccessControlFilter(routeLocator, jHipsterProperties);
        }
    }

}
```

DatabaseConfiguration

```
package org.events.salvatory.config;

import io.github.jhipster.config.JHipsterConstants;
import io.github.jhipster.config.h2.H2ConfigurationHelper;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

import org.springframework.core.env.Environment;
import org.springframework.data.jpa.repository.config.EnableJpaAuditing;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import java.sql.SQLException;

@Configuration
@EnableJpaRepositories("org.events.salvatory.repository")
@EnableJpaAuditing(auditorAwareRef = "springSecurityAuditorAware")
@EnableTransactionManagement
public class DatabaseConfiguration {

    private final Logger log = LoggerFactory.getLogger(DatabaseConfiguration.class);

    private final Environment env;

    public DatabaseConfiguration(Environment env) {
        this.env = env;
    }

    /**
     * Open the TCP port for the H2 database, so it is available remotely.
     *
     * @return the H2 database TCP server.
     * @throws SQLException if the server failed to start.
     */
    @Bean(initMethod = "start", destroyMethod = "stop")
```

```
    @Profile(JHipsterConstants.SPRING_PROFILE_DEVELOPMENT)
    public Object h2TCPServer() throws SQLException {
        String port = getValidPortForH2();
        log.debug("H2 database is available on port {}", port);
        return H2ConfigurationHelper.createServer(port);
    }

    private String getValidPortForH2() {
        int port = Integer.parseInt(env.getProperty("server.port"));
        if (port < 10000) {
            port = 10000 + port;
        } else {
            if (port < 63536) {
                port = port + 2000;
            } else {
                port = port - 2000;
            }
        }
        return String.valueOf(port);
    }
}
```

OAuth2Properties

```
package org.events.salvatory.config.oauth2;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

/**
 * OAuth2 properties define properties for OAuth2-based microservices.
 */
@Component
@ConfigurationProperties(prefix = "oauth2", ignoreUnknownFields = false)
public class OAuth2Properties {
    private WebClientConfiguration webClientConfiguration = new WebClientConfiguration();

    private SignatureVerification signatureVerification = new SignatureVerification();

    public WebClientConfiguration getWebClientConfiguration() {
        return webClientConfiguration;
    }

    public SignatureVerification getSignatureVerification() {
        return signatureVerification;
    }

    public static class WebClientConfiguration {
        private String clientId = "web_app";
        private String secret = "changeit";
        /**
         * Holds the session timeout in seconds for non-remember-me sessions.
         * After so many seconds of inactivity, the session will be terminated.
         * Only checked during token refresh, so long access token validity may
         * delay the session timeout accordingly.
         */
        private int sessionTimeoutInSeconds = 1800;
        /**
         * Defines the cookie domain. If specified, cookies will be set on this domain.
         * If not configured, then cookies will be set on the top-level domain of the
         * request you sent, i.e. if you send a request to {@code app1.your-domain.com},
         * then cookies will be set on {@code .your-domain.com}, such that they
         * are also valid for {@code app2.your-domain.com}.
         */
        private String cookieDomain;

        public String getClientId() {
            return clientId;
        }
```

```
    public void setClientId(String clientId) {
        this.clientId = clientId;
    }

    public String getSecret() {
        return secret;
    }

    public void setSecret(String secret) {
        this.secret = secret;
    }

    public int getSessionTimeoutInSeconds() {
        return sessionTimeoutInSeconds;
    }

    public void setSessionTimeoutInSeconds(int sessionTimeoutInSeconds) {
        this.sessionTimeoutInSeconds = sessionTimeoutInSeconds;
    }

    public String getCookieDomain() {
        return cookieDomain;
    }

    public void setCookieDomain(String cookieDomain) {
        this.cookieDomain = cookieDomain;
    }
}

public static class SignatureVerification {
    /**
     * Maximum refresh rate for public keys in ms.
     * We won't fetch new public keys any faster than that to avoid spamming UAA in case
     * we receive a lot of "illegal" tokens.
     */
    private long publicKeyRefreshRateLimit = 10 * 1000L;
    /**
     * Maximum TTL for the public key in ms.
     * The public key will be fetched again from UAA if it gets older than that.
     * That way, we make sure that we get the newest keys always in case they are updated there.
     */
    private long ttl = 24 * 60 * 60 * 1000L;
    /**
     * Endpoint where to retrieve the public key used to verify token signatures.
     */
    private String publicKeyEndpointUri = "http://uaa/oauth/token_key";

    public long getPublicKeyRefreshRateLimit() {
        return publicKeyRefreshRateLimit;
    }

    public void setPublicKeyRefreshRateLimit(long publicKeyRefreshRateLimit) {
        this.publicKeyRefreshRateLimit = publicKeyRefreshRateLimit;
    }

    public long getTtl() {
        return ttl;
    }

    public void setTtl(long ttl) {
        this.ttl = ttl;
    }

    public String getPublicKeyEndpointUri() {
        return publicKeyEndpointUri;
    }
```

```
        public void setPublicKeyEndpointUri(String publicKeyEndpointUri) {
            this.publicKeyEndpointUri = publicKeyEndpointUri;
        }
    }
}
```

OAuth2JwtAccessTokenConverter

```
package org.events.salvatory.config.oauth2;

import org.events.salvatory.security.oauth2.OAuth2SignatureVerifierClient;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.jwt.crypto.sign.SignatureVerifier;
import org.springframework.security.oauth2.common.exceptions.InvalidTokenException;
import org.springframework.security.oauth2.provider.token.store.JwtAccessTokenConverter;
import org.springframework.security.oauth2.provider.OAuth2Authentication;

import java.util.Map;

/**
 * Improved {@link JwtAccessTokenConverter} that can handle lazy fetching of public verifier keys.
 */
public class OAuth2JwtAccessTokenConverter extends JwtAccessTokenConverter {
    private final Logger log = LoggerFactory.getLogger(OAuth2JwtAccessTokenConverter.class);

    private final OAuth2Properties oAuth2Properties;
    private final OAuth2SignatureVerifierClient signatureVerifierClient;
    /**
     * When did we last fetch the public key?
     */
    private long lastKeyFetchTimestamp;

    public OAuth2JwtAccessTokenConverter(OAuth2Properties oAuth2Properties, OAuth2SignatureVerifierClient
signatureVerifierClient) {
        this.oAuth2Properties = oAuth2Properties;
        this.signatureVerifierClient = signatureVerifierClient;
        tryCreateSignatureVerifier();
    }

    /**
     * Try to decode the token with the current public key.
     * If it fails, contact the OAuth2 server to get a new public key, then try again.
     * We might not have fetched it in the first place or it might have changed.
     *
     * @param token the JWT token to decode.
     * @return the resulting claims.
     * @throws InvalidTokenException if we cannot decode the token.
     */
    @Override
    protected Map<String, Object> decode(String token) {
        try {
            //check if our public key and thus SignatureVerifier have expired
            long ttl = oAuth2Properties.getSignatureVerification().getTtl();
            if (ttl > 0 && System.currentTimeMillis() - lastKeyFetchTimestamp > ttl) {
                throw new InvalidTokenException("public key expired");
            }
            return super.decode(token);
        } catch (InvalidTokenException ex) {
            if (tryCreateSignatureVerifier()) {
                return super.decode(token);
            }
            throw ex;
        }
    }

    /**
     * Fetch a new public key from the AuthorizationServer.
```

```
     *
     * @return true, if we could fetch it; false, if we could not.
     */
    private boolean tryCreateSignatureVerifier() {
        long t = System.currentTimeMillis();
        if (t - lastKeyFetchTimestamp < oAuth2Properties.getSignatureVerification().getPublicKeyRefreshRateLimit()) {
            return false;
        }
        try {
            SignatureVerifier verifier = signatureVerifierClient.getSignatureVerifier();
            if (verifier != null) {
                setVerifier(verifier);
                lastKeyFetchTimestamp = t;
                log.debug("Public key retrieved from OAuth2 server to create SignatureVerifier");
                return true;
            }
        } catch (Throwable ex) {
            log.error("could not get public key from OAuth2 server to create SignatureVerifier", ex);
        }
        return false;
    }
    /**
     * Extract JWT claims and set it to OAuth2Authentication decoded details.
     * Here is how to get details:
     *
     * <pre>
     * <code>
     *  SecurityContext securityContext = SecurityContextHolder.getContext();
     *  Authentication authentication = securityContext.getAuthentication();
     *  if (authentication != null) {
     *      Object details = authentication.getDetails();
     *      if (details instanceof OAuth2AuthenticationDetails) {
     *          Object decodedDetails = ((OAuth2AuthenticationDetails) details).getDecodedDetails();
     *          if (decodedDetails != null &amp;amp;&amp;amp; decodedDetails instanceof Map) {
     *              String detailFoo = ((Map) decodedDetails).get("foo");
     *          }
     *      }
     *  }
     * </code>
     *  </pre>
     * @param claims OAuth2JWTToken claims.
     * @return {@link OAuth2Authentication}.
     */
    @Override
    public OAuth2Authentication extractAuthentication(Map<String, ?> claims) {
        OAuth2Authentication authentication = super.extractAuthentication(claims);
        authentication.setDetails(claims);
        return authentication;
    }
}
```

OAuth2AuthenticationConfiguration

```
package org.events.salvatory.config.oauth2;

import org.events.salvatory.security.oauth2.CookieTokenExtractor;
import org.events.salvatory.security.oauth2.OAuth2AuthenticationService;
import org.events.salvatory.security.oauth2.OAuth2CookieHelper;
import org.events.salvatory.security.oauth2.OAuth2TokenEndpointClient;
import org.events.salvatory.web.filter.RefreshTokenFilterConfigurer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer;
import org.springframework.security.oauth2.config.annotation.web.configuration.ResourceServerConfigurerAdapter;
import org.springframework.security.oauth2.config.annotation.web.configurers.ResourceServerSecurityConfigurer;
import org.springframework.security.oauth2.provider.authentication.TokenExtractor;
import org.springframework.security.oauth2.provider.token.TokenStore;
```

```
/**
 * Configures the RefreshFilter refreshing expired OAuth2 token Cookies.
 */
@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)
public class OAuth2AuthenticationConfiguration extends ResourceServerConfigurerAdapter {
    private final OAuth2Properties oAuth2Properties;
    private final OAuth2TokenEndpointClient tokenEndpointClient;
    private final TokenStore tokenStore;

    public OAuth2AuthenticationConfiguration(OAuth2Properties oAuth2Properties, OAuth2TokenEndpointClient
tokenEndpointClient, TokenStore tokenStore) {
        this.oAuth2Properties = oAuth2Properties;
        this.tokenEndpointClient = tokenEndpointClient;
        this.tokenStore = tokenStore;
    }

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/auth/login").permitAll()
            .antMatchers("/auth/logout").authenticated()
            .and()
            .apply(refreshTokenSecurityConfigurerAdapter());
    }

    /**
     * A {@code SecurityConfigurerAdapter} to install a servlet filter that refreshes OAuth2 tokens.
     */
    private RefreshTokenFilterConfigurer refreshTokenSecurityConfigurerAdapter() {
        return new RefreshTokenFilterConfigurer(uaaAuthenticationService(), tokenStore);
    }

    @Bean
    public OAuth2CookieHelper cookieHelper() {
        return new OAuth2CookieHelper(oAuth2Properties);
    }

    @Bean
    public OAuth2AuthenticationService uaaAuthenticationService() {
        return new OAuth2AuthenticationService(tokenEndpointClient, cookieHelper());
    }

    /**
     * Configure the ResourceServer security by installing a new {@link TokenExtractor}.
     */
    @Override
    public void configure(ResourceServerSecurityConfigurer resources) throws Exception {
        resources.tokenExtractor(tokenExtractor());
    }

    /**
     * The new {@link TokenExtractor} can extract tokens from Cookies and Authorization headers.
     *
     * @return the {@link CookieTokenExtractor} bean.
     */
    @Bean
    public TokenExtractor tokenExtractor() {
        return new CookieTokenExtractor();
    }

}
```

application.yml

```
# ===================================================================
# Spring Boot configuration.
#
# This configuration will be overridden by the Spring profile you use,
# for example application-dev.yml if you use the "dev" profile.
#
# More information on profiles: https://www.jhipster.tech/profiles/
# More information on configuration properties: https://www.jhipster.tech/common-application-properties/
# ===================================================================

# ===================================================================
# Standard Spring Boot properties.
# Full reference is available at:
# http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html
# ===================================================================

eureka:
  client:
    enabled: true
    healthcheck:
      enabled: true
    fetch-registry: true
    register-with-eureka: true
    instance-info-replication-interval-seconds: 10
    registry-fetch-interval-seconds: 10
  instance:
    appname: events_slavatory_gateway
    instanceId: events_slavatory_gateway:${spring.application.instance-id:${random.value}}
    lease-renewal-interval-in-seconds: 5
    lease-expiration-duration-in-seconds: 10
    status-page-url-path: ${management.endpoints.web.base-path}/info
    health-check-url-path: ${management.endpoints.web.base-path}/health
ribbon:
  eureka:
    enabled: true
zuul:
  sensitive-headers: Cookie,Set-Cookie
  host:
    max-total-connections: 1000
    max-per-route-connections: 100
  prefix: /services
  semaphore:
    max-semaphores: 500

hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 10000

management:
  endpoints:
    web:
      base-path: /management
      exposure:
        include: ['configprops', 'env', 'health', 'info', 'jhimetrics', 'logfile', 'loggers', 'prometheus', 'threaddump']
  endpoint:
    health:
      show-details: when-authorized
      roles: 'ROLE_ADMIN'
    jhimetrics:
      enabled: true
  info:
    git:
      mode: full
```

```
  health:
    mail:
      enabled: false # When using the MailService, configure an SMTP server and set this to true
  metrics:
    export:
      # Prometheus is the default metrics backend
      prometheus:
        enabled: true
        step: 60
    enable:
      http: true
      jvm: true
      logback: true
      process: true
      system: true
    distribution:
      percentiles-histogram:
        all: true
      percentiles:
        all: 0, 0.5, 0.75, 0.95, 0.99, 1.0
    tags:
      application: ${spring.application.name}
    web:
      server:
        auto-time-requests: true

spring:
  application:
    name: events_slavatory_gateway
  jmx:
    enabled: false
  data:
    jpa:
      repositories:
        bootstrap-mode: deferred
  jpa:
    open-in-view: false
    properties:
      hibernate.jdbc.time_zone: UTC
    hibernate:
      ddl-auto: none
      naming:
        physical-strategy: org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNamingStrategy
        implicit-strategy: org.springframework.boot.orm.jpa.hibernate.SpringImplicitNamingStrategy
  messages:
    basename: i18n/messages
  main:
    allow-bean-definition-overriding: true
  mvc:
    favicon:
      enabled: false
  task:
    execution:
      thread-name-prefix: events-slavatory-gateway-task-
      pool:
        core-size: 2
        max-size: 50
        queue-capacity: 10000
    scheduling:
      thread-name-prefix: events-slavatory-gateway-scheduling-
      pool:
        size: 2
  thymeleaf:
    mode: HTML
  output:
    ansi:
      console-available: true
```

```
security:
  oauth2:
    client:
      access-token-uri: http://localhost:9080/auth/realms/jhipster/protocol/openid-connect/token
      user-authorization-uri: http://localhost:9080/auth/realms/jhipster/protocol/openid-connect/auth
      client-id: web_app
      client-secret: web_app
      scope: openid profile email
    resource:
      filter-order: 3
      user-info-uri: http://localhost:9080/auth/realms/jhipster/protocol/openid-connect/userinfo

server:
  servlet:
    session:
      cookie:
        http-only: true

# Properties to be exposed on the /info management endpoint
info:
  # Comma separated list of profiles that will trigger the ribbon to show
  display-ribbon-on-profiles: 'dev'

# ===================================================================
# JHipster specific properties
#
# Full reference is available at: https://www.jhipster.tech/common-application-properties/
# ===================================================================

jhipster:
  clientApp:
    name: 'eventsSlavatoryGatewayApp'
  # By default CORS is disabled. Uncomment to enable.
  # cors:
  #     allowed-origins: "*"
  #     allowed-methods: "*"
  #     allowed-headers: "*"
  #     exposed-headers: "Authorization,Link,X-Total-Count"
  #     allow-credentials: true
  #     max-age: 1800
  mail:
    from: events_slavatory_gateway@localhost
  swagger:
    default-include-pattern: /api/.*
    title: events_slavatory_gateway API
    description: events_slavatory_gateway API documentation
    version: 0.0.1
    terms-of-service-url:
    contact-name:
    contact-url:
    contact-email:
    license:
    license-url:
# ===================================================================
# Application specific properties
# Add your own application properties here, see the ApplicationProperties class
# to have type-safe configuration, like in the JHipsterProperties above
#
# More documentation is available at:
# https://www.jhipster.tech/common-application-properties/
# ===================================================================

# application:
EventsSalvatoryEventsServiceApp
package org.events.salvatory;

import org.events.salvatory.client.OAuth2InterceptedFeignConfiguration;
```

```
import org.events.salvatory.config.ApplicationProperties;
import org.events.salvatory.config.DefaultProfileUtil;

import io.github.jhipster.config.JHipsterConstants;

import org.apache.commons.lang3.StringUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.liquibase.LiquibaseProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.FilterType;
import org.springframework.core.env.Environment;

import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Arrays;
import java.util.Collection;

@ComponentScan(
    excludeFilters = @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE, classes =
OAuth2InterceptedFeignConfiguration.class)
)
@SpringBootApplication
@EnableConfigurationProperties({LiquibaseProperties.class, ApplicationProperties.class})
@EnableDiscoveryClient
public class EventsSalvatoryEventsServiceApp implements InitializingBean {

    private static final Logger log = LoggerFactory.getLogger(EventsSalvatoryEventsServiceApp.class);

    private final Environment env;

    public EventsSalvatoryEventsServiceApp(Environment env) {
        this.env = env;
    }

    /**
     * Initializes eventsSalvatoryEventsService.
     * <p>
     * Spring profiles can be configured with a program argument --spring.profiles.active=your-active-profile
     * <p>
     * You can find more information on how profiles work with JHipster on <a
href="https://www.jhipster.tech/profiles/">https://www.jhipster.tech/profiles/</a>.
     */
    @Override
    public void afterPropertiesSet() throws Exception {
        Collection<String> activeProfiles = Arrays.asList(env.getActiveProfiles());
        if (activeProfiles.contains(JHipsterConstants.SPRING_PROFILE_DEVELOPMENT) &&
activeProfiles.contains(JHipsterConstants.SPRING_PROFILE_PRODUCTION)) {
            log.error("You have misconfigured your application! It should not run " +
                "with both the 'dev' and 'prod' profiles at the same time.");
        }
        if (activeProfiles.contains(JHipsterConstants.SPRING_PROFILE_DEVELOPMENT) &&
activeProfiles.contains(JHipsterConstants.SPRING_PROFILE_CLOUD)) {
            log.error("You have misconfigured your application! It should not " +
                "run with both the 'dev' and 'cloud' profiles at the same time.");
        }
    }

    /**
     * Main method, used to run the application.
     *
     * @param args the command line arguments.
```

```
     */
    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(EventsSalvatoryEventsServiceApp.class);
        DefaultProfileUtil.addDefaultProfile(app);
        Environment env = app.run(args).getEnvironment();
        logApplicationStartup(env);
    }

    private static void logApplicationStartup(Environment env) {
        String protocol = "http";
        if (env.getProperty("server.ssl.key-store") != null) {
            protocol = "https";
        }
        String serverPort = env.getProperty("server.port");
        String contextPath = env.getProperty("server.servlet.context-path");
        if (StringUtils.isBlank(contextPath)) {
            contextPath = "/";
        }
        String hostAddress = "localhost";
        try {
            hostAddress = InetAddress.getLocalHost().getHostAddress();
        } catch (UnknownHostException e) {
            log.warn("The host name could not be determined, using `localhost` as fallback");
        }
        log.info("\n----------------------------------------------------------\n\t" +
                "Application '{}' is running! Access URLs:\n\t" +
                "Local: \t\t{}://localhost:{}{}\n\t" +
                "External: \t{}://{}:{}{}\n\t" +
                "Profile(s): \t{}\n----------------------------------------------------------",
            env.getProperty("spring.application.name"),
            protocol,
            serverPort,
            contextPath,
            protocol,
            hostAddress,
            serverPort,
            contextPath,
            env.getActiveProfiles());

        String configServerStatus = env.getProperty("configserver.status");
        if (configServerStatus == null) {
            configServerStatus = "Not found or not setup for this application";
        }
        log.info("\n----------------------------------------------------------\n\t" +
                "Config Server: \t{}\n----------------------------------------------------------", configServerStatus);
    }
}
```

OrganizationResource

```
package org.events.salvatory.web.rest;

import org.events.salvatory.domain.Organization;
import org.events.salvatory.repository.OrganizationRepository;
import org.events.salvatory.web.rest.errors.BadRequestAlertException;

import io.github.jhipster.web.util.HeaderUtil;
import io.github.jhipster.web.util.ResponseUtil;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.ResponseEntity;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.*;

import java.net.URI;
import java.net.URISyntaxException;

import java.util.List;
```

```
import java.util.Optional;

/**
 * REST controller for managing {@link org.events.salvatory.domain.Organization}.
 */
@RestController
@RequestMapping("/api")
@Transactional
public class OrganizationResource {

    private final Logger log = LoggerFactory.getLogger(OrganizationResource.class);

    private static final String ENTITY_NAME = "eventsSalvatoryEventsServiceOrganization";

    @Value("${jhipster.clientApp.name}")
    private String applicationName;

    private final OrganizationRepository organizationRepository;

    public OrganizationResource(OrganizationRepository organizationRepository) {
        this.organizationRepository = organizationRepository;
    }

    /**
     * {@code POST  /organizations} : Create a new organization.
     *
     * @param organization the organization to create.
     * @return the {@link ResponseEntity} with status {@code 201 (Created)} and with body the new organization, or with
status {@code 400 (Bad Request)} if the organization has already an ID.
     * @throws URISyntaxException if the Location URI syntax is incorrect.
     */
    @PostMapping("/organizations")
    public ResponseEntity<Organization> createOrganization(@RequestBody Organization organization) throws
URISyntaxException {
        log.debug("REST request to save Organization : {}", organization);
        if (organization.getId() != null) {
            throw new BadRequestAlertException("A new organization cannot already have an ID", ENTITY_NAME,
"idexists");
        }
        Organization result = organizationRepository.save(organization);
        return ResponseEntity.created(new URI("/api/organizations/" + result.getId()))
            .headers(HeaderUtil.createEntityCreationAlert(applicationName, false, ENTITY_NAME, result.getId().toString()))
            .body(result);
    }

    /**
     * {@code PUT  /organizations} : Updates an existing organization.
     *
     * @param organization the organization to update.
     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and with body the updated organization,
     * or with status {@code 400 (Bad Request)} if the organization is not valid,
     * or with status {@code 500 (Internal Server Error)} if the organization couldn't be updated.
     * @throws URISyntaxException if the Location URI syntax is incorrect.
     */
    @PutMapping("/organizations")
    public ResponseEntity<Organization> updateOrganization(@RequestBody Organization organization) throws
URISyntaxException {
        log.debug("REST request to update Organization : {}", organization);
        if (organization.getId() == null) {
            throw new BadRequestAlertException("Invalid id", ENTITY_NAME, "idnull");
        }
        Organization result = organizationRepository.save(organization);
        return ResponseEntity.ok()
            .headers(HeaderUtil.createEntityUpdateAlert(applicationName, false, ENTITY_NAME,
organization.getId().toString()))
            .body(result);
    }
```

```
    /**
     * {@code GET  /organizations} : get all the organizations.
     *

     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and the list of organizations in body.
     */
    @GetMapping("/organizations")
    public List<Organization> getAllOrganizations() {
        log.debug("REST request to get all Organizations");
        return organizationRepository.findAll();
    }

    /**
     * {@code GET  /organizations/:id} : get the "id" organization.
     *
     * @param id the id of the organization to retrieve.
     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and with body the organization, or with status
{@code 404 (Not Found)}.
     */
    @GetMapping("/organizations/{id}")
    public ResponseEntity<Organization> getOrganization(@PathVariable Long id) {
        log.debug("REST request to get Organization : {}", id);
        Optional<Organization> organization = organizationRepository.findById(id);
        return ResponseUtil.wrapOrNotFound(organization);
    }

    /**
     * {@code DELETE  /organizations/:id} : delete the "id" organization.
     *
     * @param id the id of the organization to delete.
     * @return the {@link ResponseEntity} with status {@code 204 (NO_CONTENT)}.
     */
    @DeleteMapping("/organizations/{id}")
    public ResponseEntity<Void> deleteOrganization(@PathVariable Long id) {
        log.debug("REST request to delete Organization : {}", id);
        organizationRepository.deleteById(id);
        return ResponseEntity.noContent().headers(HeaderUtil.createEntityDeletionAlert(applicationName, false,
ENTITY_NAME, id.toString())).build();
    }
}
```

LocationResource

```
package org.events.salvatory.web.rest;

import org.events.salvatory.domain.Location;
import org.events.salvatory.repository.LocationRepository;
import org.events.salvatory.web.rest.errors.BadRequestAlertException;

import io.github.jhipster.web.util.HeaderUtil;
import io.github.jhipster.web.util.ResponseUtil;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.ResponseEntity;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.*;

import java.net.URI;
import java.net.URISyntaxException;

import java.util.List;
import java.util.Optional;

/**
 * REST controller for managing {@link org.events.salvatory.domain.Location}.
 */
@RestController
```

```
@RequestMapping("/api")
@Transactional
public class LocationResource {

    private final Logger log = LoggerFactory.getLogger(LocationResource.class);

    private static final String ENTITY_NAME = "eventsSalvatoryEventsServiceLocation";

    @Value("${jhipster.clientApp.name}")
    private String applicationName;

    private final LocationRepository locationRepository;

    public LocationResource(LocationRepository locationRepository) {
        this.locationRepository = locationRepository;
    }

    /**
     * {@code POST  /locations} : Create a new location.
     *
     * @param location the location to create.
     * @return the {@link ResponseEntity} with status {@code 201 (Created)} and with body the new location, or with status
{@code 400 (Bad Request)} if the location has already an ID.
     * @throws URISyntaxException if the Location URI syntax is incorrect.
     */
    @PostMapping("/locations")
    public ResponseEntity<Location> createLocation(@RequestBody Location location) throws URISyntaxException {
        log.debug("REST request to save Location : {}", location);
        if (location.getId() != null) {
            throw new BadRequestAlertException("A new location cannot already have an ID", ENTITY_NAME, "idexists");
        }
        Location result = locationRepository.save(location);
        return ResponseEntity.created(new URI("/api/locations/" + result.getId()))
            .headers(HeaderUtil.createEntityCreationAlert(applicationName, false, ENTITY_NAME, result.getId().toString()))
            .body(result);
    }

    /**
     * {@code PUT  /locations} : Updates an existing location.
     *
     * @param location the location to update.
     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and with body the updated location,
     * or with status {@code 400 (Bad Request)} if the location is not valid,
     * or with status {@code 500 (Internal Server Error)} if the location couldn't be updated.
     * @throws URISyntaxException if the Location URI syntax is incorrect.
     */
    @PutMapping("/locations")
    public ResponseEntity<Location> updateLocation(@RequestBody Location location) throws URISyntaxException {
        log.debug("REST request to update Location : {}", location);
        if (location.getId() == null) {
            throw new BadRequestAlertException("Invalid id", ENTITY_NAME, "idnull");
        }
        Location result = locationRepository.save(location);
        return ResponseEntity.ok()
            .headers(HeaderUtil.createEntityUpdateAlert(applicationName, false, ENTITY_NAME, location.getId().toString()))
            .body(result);
    }

    /**
     * {@code GET  /locations} : get all the locations.
     *

     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and the list of locations in body.
     */
    @GetMapping("/locations")
    public List<Location> getAllLocations() {
        log.debug("REST request to get all Locations");
```

```
        return locationRepository.findAll();
    }

    /**
     * {@code GET  /locations/:id} : get the "id" location.
     *
     * @param id the id of the location to retrieve.
     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and with body the location, or with status {@code
404 (Not Found)}.
     */
    @GetMapping("/locations/{id}")
    public ResponseEntity<Location> getLocation(@PathVariable Long id) {
        log.debug("REST request to get Location : {}", id);
        Optional<Location> location = locationRepository.findById(id);
        return ResponseUtil.wrapOrNotFound(location);
    }

    /**
     * {@code DELETE  /locations/:id} : delete the "id" location.
     *
     * @param id the id of the location to delete.
     * @return the {@link ResponseEntity} with status {@code 204 (NO_CONTENT)}.
     */
    @DeleteMapping("/locations/{id}")
    public ResponseEntity<Void> deleteLocation(@PathVariable Long id) {
        log.debug("REST request to delete Location : {}", id);
        locationRepository.deleteById(id);
        return ResponseEntity.noContent().headers(HeaderUtil.createEntityDeletionAlert(applicationName, false,
ENTITY_NAME, id.toString())).build();
    }
}
```

EventResource

```
package org.events.salvatory.web.rest;

import org.events.salvatory.domain.Event;
import org.events.salvatory.repository.EventRepository;
import org.events.salvatory.web.rest.errors.BadRequestAlertException;

import io.github.jhipster.web.util.HeaderUtil;
import io.github.jhipster.web.util.ResponseUtil;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.ResponseEntity;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.*;

import java.net.URI;
import java.net.URISyntaxException;

import java.util.List;
import java.util.Optional;

/**
 * REST controller for managing {@link org.events.salvatory.domain.Event}.
 */
@RestController
@RequestMapping("/api")
@Transactional
public class EventResource {

    private final Logger log = LoggerFactory.getLogger(EventResource.class);

    private static final String ENTITY_NAME = "eventsSalvatoryEventsServiceEvent";

    @Value("${jhipster.clientApp.name}")
    private String applicationName;
```

```
    private final EventRepository eventRepository;

    public EventResource(EventRepository eventRepository) {
        this.eventRepository = eventRepository;
    }

    /**
     * {@code POST  /events} : Create a new event.
     *
     * @param event the event to create.
     * @return the {@link ResponseEntity} with status {@code 201 (Created)} and with body the new event, or with status
{@code 400 (Bad Request)} if the event has already an ID.
     * @throws URISyntaxException if the Location URI syntax is incorrect.
     */
    @PostMapping("/events")
    public ResponseEntity<Event> createEvent(@RequestBody Event event) throws URISyntaxException {
        log.debug("REST request to save Event : {}", event);
        if (event.getId() != null) {
            throw new BadRequestAlertException("A new event cannot already have an ID", ENTITY_NAME, "idexists");
        }
        Event result = eventRepository.save(event);
        return ResponseEntity.created(new URI("/api/events/" + result.getId()))
            .headers(HeaderUtil.createEntityCreationAlert(applicationName, false, ENTITY_NAME, result.getId().toString()))
            .body(result);
    }

    /**
     * {@code PUT  /events} : Updates an existing event.
     *
     * @param event the event to update.
     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and with body the updated event,
     * or with status {@code 400 (Bad Request)} if the event is not valid,
     * or with status {@code 500 (Internal Server Error)} if the event couldn't be updated.
     * @throws URISyntaxException if the Location URI syntax is incorrect.
     */
    @PutMapping("/events")
    public ResponseEntity<Event> updateEvent(@RequestBody Event event) throws URISyntaxException {
        log.debug("REST request to update Event : {}", event);
        if (event.getId() == null) {
            throw new BadRequestAlertException("Invalid id", ENTITY_NAME, "idnull");
        }
        Event result = eventRepository.save(event);
        return ResponseEntity.ok()
            .headers(HeaderUtil.createEntityUpdateAlert(applicationName, false, ENTITY_NAME, event.getId().toString()))
            .body(result);
    }

    /**
     * {@code GET  /events} : get all the events.
     *

     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and the list of events in body.
     */
    @GetMapping("/events")
    public List<Event> getAllEvents() {
        log.debug("REST request to get all Events");
        return eventRepository.findAll();
    }

    /**
     * {@code GET  /events/:id} : get the "id" event.
     *
     * @param id the id of the event to retrieve.
     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and with body the event, or with status {@code 404
(Not Found)}.
     */
```

```
    @GetMapping("/events/{id}")
    public ResponseEntity<Event> getEvent(@PathVariable Long id) {
        log.debug("REST request to get Event : {}", id);
        Optional<Event> event = eventRepository.findById(id);
        return ResponseUtil.wrapOrNotFound(event);
    }

    /**
     * {@code DELETE  /events/:id} : delete the "id" event.
     *
     * @param id the id of the event to delete.
     * @return the {@link ResponseEntity} with status {@code 204 (NO_CONTENT)}.
     */
    @DeleteMapping("/events/{id}")
    public ResponseEntity<Void> deleteEvent(@PathVariable Long id) {
        log.debug("REST request to delete Event : {}", id);
        eventRepository.deleteById(id);
        return ResponseEntity.noContent().headers(HeaderUtil.createEntityDeletionAlert(applicationName, false,
ENTITY_NAME, id.toString())).build();
    }
}
UaaSignatureVerifierClient
package org.events.salvatory.security.oauth2;

import org.events.salvatory.config.oauth2.OAuth2Properties;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.security.jwt.crypto.sign.RsaVerifier;
import org.springframework.security.jwt.crypto.sign.SignatureVerifier;
import org.springframework.security.oauth2.common.exceptions.InvalidClientException;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import java.util.Map;

/**
 * Client fetching the public key from UAA to create a {@link SignatureVerifier}.
 */
@Component
public class UaaSignatureVerifierClient implements OAuth2SignatureVerifierClient {
    private final Logger log = LoggerFactory.getLogger(UaaSignatureVerifierClient.class);
    private final RestTemplate restTemplate;
    protected final OAuth2Properties oAuth2Properties;

    public UaaSignatureVerifierClient(DiscoveryClient discoveryClient, @Qualifier("loadBalancedRestTemplate")
RestTemplate restTemplate,
                                      OAuth2Properties oAuth2Properties) {
        this.restTemplate = restTemplate;
        this.oAuth2Properties = oAuth2Properties;
        // Load available UAA servers
        discoveryClient.getServices();
    }

    /**
     * Fetches the public key from the UAA.
     *
     * @return the public key used to verify JWT tokens; or {@code null}.
     */
    @Override
    public SignatureVerifier getSignatureVerifier() throws Exception {
        try {
            HttpEntity<Void> request = new HttpEntity<Void>(new HttpHeaders());
```

```
            String key = (String) restTemplate
                .exchange(getPublicKeyEndpoint(), HttpMethod.GET, request, Map.class).getBody()
                .get("value");
            return new RsaVerifier(key);
        } catch (IllegalStateException ex) {
            log.warn("could not contact UAA to get public key");
            return null;
        }
    }

    /**
     * Returns the configured endpoint URI to retrieve the public key.
     *
     * @return the configured endpoint URI to retrieve the public key.
     */
    private String getPublicKeyEndpoint() {
        String tokenEndpointUrl = oAuth2Properties.getSignatureVerification().getPublicKeyEndpointUri();
        if (tokenEndpointUrl == null) {
            throw new InvalidClientException("no token endpoint configured in application properties");
        }
        return tokenEndpointUrl;
    }
}
Organization
package org.events.salvatory.domain;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

import javax.persistence.*;

import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;

/**
 * A Organization.
 */
@Entity
@Table(name = "organization")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Organization implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "description")
    private String description;

    @Column(name = "rating")
    private Long rating;

    @Column(name = "creator_name")
    private String creatorName;

    @Column(name = "creation_date")
    private String creationDate;

    @OneToMany(mappedBy = "organization")
    @Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
    private Set<Event> events = new HashSet<>();
```

```
// jhipster-needle-entity-add-field - JHipster will add fields here, do not remove
public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

public String getName() {
    return name;
}

public Organization name(String name) {
    this.name = name;
    return this;
}

public void setName(String name) {
    this.name = name;
}

public String getDescription() {
    return description;
}

public Organization description(String description) {
    this.description = description;
    return this;
}

public void setDescription(String description) {
    this.description = description;
}

public Long getRating() {
    return rating;
}

public Organization rating(Long rating) {
    this.rating = rating;
    return this;
}

public void setRating(Long rating) {
    this.rating = rating;
}

public String getCreatorName() {
    return creatorName;
}

public Organization creatorName(String creatorName) {
    this.creatorName = creatorName;
    return this;
}

public void setCreatorName(String creatorName) {
    this.creatorName = creatorName;
}

public String getCreationDate() {
    return creationDate;
}

public Organization creationDate(String creationDate) {
    this.creationDate = creationDate;
```

```
        return this;
    }

    public void setCreationDate(String creationDate) {
        this.creationDate = creationDate;
    }

    public Set<Event> getEvents() {
        return events;
    }

    public Organization events(Set<Event> events) {
        this.events = events;
        return this;
    }

    public Organization addEvents(Event event) {
        this.events.add(event);
        event.setOrganization(this);
        return this;
    }

    public Organization removeEvents(Event event) {
        this.events.remove(event);
        event.setOrganization(null);
        return this;
    }

    public void setEvents(Set<Event> events) {
        this.events = events;
    }
    // jhipster-needle-entity-add-getters-setters - JHipster will add getters and setters here, do not remove

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Organization)) {
            return false;
        }
        return id != null && id.equals(((Organization) o).id);
    }

    @Override
    public int hashCode() {
        return 31;
    }

    @Override
    public String toString() {
        return "Organization{" +
            "id=" + getId() +
            ", name='" + getName() + "'" +
            ", description='" + getDescription() + "'" +
            ", rating=" + getRating() +
            ", creatorName='" + getCreatorName() + "'" +
            ", creationDate='" + getCreationDate() + "'" +
            "}";
    }
}
Location
package org.events.salvatory.domain;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

import javax.persistence.*;
```

```
import java.io.Serializable;

/**
 * A Location.
 */
@Entity
@Table(name = "location")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Location implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "street_address")
    private String streetAddress;

    @Column(name = "postal_code")
    private String postalCode;

    @Column(name = "city")
    private String city;

    @Column(name = "state_province")
    private String stateProvince;

    // jhipster-needle-entity-add-field - JHipster will add fields here, do not remove
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getStreetAddress() {
        return streetAddress;
    }

    public Location streetAddress(String streetAddress) {
        this.streetAddress = streetAddress;
        return this;
    }

    public void setStreetAddress(String streetAddress) {
        this.streetAddress = streetAddress;
    }

    public String getPostalCode() {
        return postalCode;
    }

    public Location postalCode(String postalCode) {
        this.postalCode = postalCode;
        return this;
    }

    public void setPostalCode(String postalCode) {
        this.postalCode = postalCode;
    }

    public String getCity() {
        return city;
    }
```

```
    public Location city(String city) {
        this.city = city;
        return this;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getStateProvince() {
        return stateProvince;
    }

    public Location stateProvince(String stateProvince) {
        this.stateProvince = stateProvince;
        return this;
    }

    public void setStateProvince(String stateProvince) {
        this.stateProvince = stateProvince;
    }
    // jhipster-needle-entity-add-getters-setters - JHipster will add getters and setters here, do not remove

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Location)) {
            return false;
        }
        return id != null && id.equals(((Location) o).id);
    }

    @Override
    public int hashCode() {
        return 31;
    }

    @Override
    public String toString() {
        return "Location{" +
            "id=" + getId() +
            ", streetAddress='" + getStreetAddress() + "'" +
            ", postalCode='" + getPostalCode() + "'" +
            ", city='" + getCity() + "'" +
            ", stateProvince='" + getStateProvince() + "'" +
            "}";
    }
}
```

Event

```
package org.events.salvatory.domain;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

import javax.persistence.*;

import java.io.Serializable;
import java.time.Instant;

import org.events.salvatory.domain.enumeration.EventType;

/**
 * A Event.
 */
```

```
@Entity
@Table(name = "event")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Event implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "description")
    private String description;

    @Column(name = "price")
    private Long price;

    @Column(name = "address")
    private String address;

    @Column(name = "image")
    private String image;

    @Column(name = "rating")
    private Long rating;

    @Enumerated(EnumType.STRING)
    @Column(name = "event_type")
    private EventType eventType;

    @Column(name = "creation_date")
    private Instant creationDate;

    @Column(name = "start_date")
    private Instant startDate;

    @Column(name = "end_date")
    private Instant endDate;

    @OneToOne(fetch = FetchType.LAZY)

    @JoinColumn(unique = true)
    private Location location;

    @ManyToOne(fetch = FetchType.LAZY)
    @JsonIgnoreProperties("events")
    private Organization organization;

    // jhipster-needle-entity-add-field - JHipster will add fields here, do not remove
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public Event name(String name) {
        this.name = name;
        return this;
```

```
}

public void setName(String name) {
    this.name = name;
}

public String getDescription() {
    return description;
}

public Event description(String description) {
    this.description = description;
    return this;
}

public void setDescription(String description) {
    this.description = description;
}

public Long getPrice() {
    return price;
}

public Event price(Long price) {
    this.price = price;
    return this;
}

public void setPrice(Long price) {
    this.price = price;
}

public String getAddress() {
    return address;
}

public Event address(String address) {
    this.address = address;
    return this;
}

public void setAddress(String address) {
    this.address = address;
}

public String getImage() {
    return image;
}

public Event image(String image) {
    this.image = image;
    return this;
}

public void setImage(String image) {
    this.image = image;
}

public Long getRating() {
    return rating;
}

public Event rating(Long rating) {
    this.rating = rating;
    return this;
}
```

```
public void setRating(Long rating) {
    this.rating = rating;
}

public EventType getEventType() {
    return eventType;
}

public Event eventType(EventType eventType) {
    this.eventType = eventType;
    return this;
}

public void setEventType(EventType eventType) {
    this.eventType = eventType;
}

public Instant getCreationDate() {
    return creationDate;
}

public Event creationDate(Instant creationDate) {
    this.creationDate = creationDate;
    return this;
}

public void setCreationDate(Instant creationDate) {
    this.creationDate = creationDate;
}

public Instant getStartDate() {
    return startDate;
}

public Event startDate(Instant startDate) {
    this.startDate = startDate;
    return this;
}

public void setStartDate(Instant startDate) {
    this.startDate = startDate;
}

public Instant getEndDate() {
    return endDate;
}

public Event endDate(Instant endDate) {
    this.endDate = endDate;
    return this;
}

public void setEndDate(Instant endDate) {
    this.endDate = endDate;
}

public Location getLocation() {
    return location;
}

public Event location(Location location) {
    this.location = location;
    return this;
}

public void setLocation(Location location) {
    this.location = location;
```

```
    }

    public Organization getOrganization() {
        return organization;
    }

    public Event organization(Organization organization) {
        this.organization = organization;
        return this;
    }

    public void setOrganization(Organization organization) {
        this.organization = organization;
    }
    // jhipster-needle-entity-add-getters-setters - JHipster will add getters and setters here, do not remove

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Event)) {
            return false;
        }
        return id != null && id.equals(((Event) o).id);
    }

    @Override
    public int hashCode() {
        return 31;
    }

    @Override
    public String toString() {
        return "Event{" +
            "id=" + getId() +
            ", name='" + getName() + "'" +
            ", description='" + getDescription() + "'" +
            ", price=" + getPrice() +
            ", address='" + getAddress() + "'" +
            ", image='" + getImage() + "'" +
            ", rating=" + getRating() +
            ", eventType='" + getEventType() + "'" +
            ", creationDate='" + getCreationDate() + "'" +
            ", startDate='" + getStartDate() + "'" +
            ", endDate='" + getEndDate() + "'" +
            "}";
    }
}
```

WebConfigurer

```
package org.events.salvatory.config;

import io.github.jhipster.config.JHipsterConstants;
import io.github.jhipster.config.JHipsterProperties;
import io.github.jhipster.config.h2.H2ConfigurationHelper;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.web.server.*;
import org.springframework.boot.web.servlet.ServletContextInitializer;
import org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.Environment;
import org.springframework.core.env.Profiles;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import org.springframework.web.filter.CorsFilter;
```

```
import javax.servlet.*;

/**
 * Configuration of web application with Servlet 3.0 APIs.
 */
@Configuration
public class WebConfigurer implements ServletContextInitializer {

    private final Logger log = LoggerFactory.getLogger(WebConfigurer.class);

    private final Environment env;

    private final JHipsterProperties jHipsterProperties;

    public WebConfigurer(Environment env, JHipsterProperties jHipsterProperties) {
        this.env = env;
        this.jHipsterProperties = jHipsterProperties;
    }

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        if (env.getActiveProfiles().length != 0) {
            log.info("Web application configuration, using profiles: {}", (Object[]) env.getActiveProfiles());
        }
        if (env.acceptsProfiles(Profiles.of(JHipsterConstants.SPRING_PROFILE_DEVELOPMENT))) {
            initH2Console(servletContext);
        }
        log.info("Web application fully configured");
    }

    @Bean
    public CorsFilter corsFilter() {
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        CorsConfiguration config = jHipsterProperties.getCors();
        if (config.getAllowedOrigins() != null && !config.getAllowedOrigins().isEmpty()) {
            log.debug("Registering CORS filter");
            source.registerCorsConfiguration("/api/**", config);
            source.registerCorsConfiguration("/management/**", config);
            source.registerCorsConfiguration("/v2/api-docs", config);
        }
        return new CorsFilter(source);
    }
SecurityConfiguration

    /**
     * Initializes H2 console.
     */
    private void initH2Console(ServletContext servletContext) {
        log.debug("Initialize H2 console");
        H2ConfigurationHelper.initH2Console(servletContext);
    }

}
package org.events.salvatory.config;

import org.events.salvatory.config.oauth2.OAuth2JwtAccessTokenConverter;
import org.events.salvatory.config.oauth2.OAuth2Properties;
import org.events.salvatory.security.oauth2.OAuth2SignatureVerifierClient;
import org.events.salvatory.security.AuthoritiesConstants;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.cloud.client.loadbalancer.RestTemplateCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
```

```
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer;
import org.springframework.security.oauth2.config.annotation.web.configuration.ResourceServerConfigurerAdapter;
import org.springframework.security.oauth2.provider.token.TokenStore;
import org.springframework.security.oauth2.provider.token.store.JwtAccessTokenConverter;
import org.springframework.security.oauth2.provider.token.store.JwtTokenStore;
import org.springframework.security.web.csrf.CookieCsrfTokenRepository;
import org.springframework.web.client.RestTemplate;

@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)
public class SecurityConfiguration extends ResourceServerConfigurerAdapter {
    private final OAuth2Properties oAuth2Properties;

    public SecurityConfiguration(OAuth2Properties oAuth2Properties) {
        this.oAuth2Properties = oAuth2Properties;
    }

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
            .csrf()
            .disable()
            .headers()
            .frameOptions()
            .disable()
        .and()
            .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and()
            .authorizeRequests()
            .antMatchers("/api/**").authenticated()
            .antMatchers("/management/health").permitAll()
            .antMatchers("/management/info").permitAll()
            .antMatchers("/management/prometheus").permitAll()
            .antMatchers("/management/**").hasAuthority(AuthoritiesConstants.ADMIN);
    }

    @Bean
    public TokenStore tokenStore(JwtAccessTokenConverter jwtAccessTokenConverter) {
        return new JwtTokenStore(jwtAccessTokenConverter);
    }

    @Bean
    public JwtAccessTokenConverter jwtAccessTokenConverter(OAuth2SignatureVerifierClient signatureVerifierClient) {
        return new OAuth2JwtAccessTokenConverter(oAuth2Properties, signatureVerifierClient);
    }

    @Bean
    @Qualifier("loadBalancedRestTemplate")
    public RestTemplate loadBalancedRestTemplate(RestTemplateCustomizer customizer) {
        RestTemplate restTemplate = new RestTemplate();
        customizer.customize(restTemplate);
        return restTemplate;
    }

    @Bean
    @Qualifier("vanillaRestTemplate")
    public RestTemplate vanillaRestTemplate() {
        return new RestTemplate();
    }
}
DatabaseConfiguration
package org.events.salvatory.config;

import io.github.jhipster.config.JHipsterConstants;
```

```
import io.github.jhipster.config.h2.H2ConfigurationHelper;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

import org.springframework.core.env.Environment;
import org.springframework.data.jpa.repository.config.EnableJpaAuditing;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import java.sql.SQLException;

@Configuration
@EnableJpaRepositories("org.events.salvatory.repository")
@EnableJpaAuditing(auditorAwareRef = "springSecurityAuditorAware")
@EnableTransactionManagement
public class DatabaseConfiguration {

    private final Logger log = LoggerFactory.getLogger(DatabaseConfiguration.class);

    private final Environment env;

    public DatabaseConfiguration(Environment env) {
        this.env = env;
    }

    /**
     * Open the TCP port for the H2 database, so it is available remotely.
     *
     * @return the H2 database TCP server.
     * @throws SQLException if the server failed to start.
     */
    @Bean(initMethod = "start", destroyMethod = "stop")
    @Profile(JHipsterConstants.SPRING_PROFILE_DEVELOPMENT)
    public Object h2TCPServer() throws SQLException {
        String port = getValidPortForH2();
        log.debug("H2 database is available on port {}", port);
        return H2ConfigurationHelper.createServer(port);
    }

    private String getValidPortForH2() {
        int port = Integer.parseInt(env.getProperty("server.port"));
        if (port < 10000) {
            port = 10000 + port;
        } else {
            if (port < 63536) {
                port = port + 2000;
            } else {
                port = port - 2000;
            }
        }
        return String.valueOf(port);
    }
}
OAuth2Properties
package org.events.salvatory.config.oauth2;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

/**
 * OAuth2 properties define properties for OAuth2-based microservices.
 */
@Component
@ConfigurationProperties(prefix = "oauth2", ignoreUnknownFields = false)
```

```
public class OAuth2Properties {
    private WebClientConfiguration webClientConfiguration = new WebClientConfiguration();

    private SignatureVerification signatureVerification = new SignatureVerification();

    public WebClientConfiguration getWebClientConfiguration() {
        return webClientConfiguration;
    }

    public SignatureVerification getSignatureVerification() {
        return signatureVerification;
    }

    public static class WebClientConfiguration {
        private String clientId = "web_app";
        private String secret = "changeit";
        /**
         * Holds the session timeout in seconds for non-remember-me sessions.
         * After so many seconds of inactivity, the session will be terminated.
         * Only checked during token refresh, so long access token validity may
         * delay the session timeout accordingly.
         */
        private int sessionTimeoutInSeconds = 1800;
        /**
         * Defines the cookie domain. If specified, cookies will be set on this domain.
         * If not configured, then cookies will be set on the top-level domain of the
         * request you sent, i.e. if you send a request to {@code app1.your-domain.com},
         * then cookies will be set on {@code .your-domain.com}, such that they
         * are also valid for {@code app2.your-domain.com}.
         */
        private String cookieDomain;

        public String getClientId() {
            return clientId;
        }

        public void setClientId(String clientId) {
            this.clientId = clientId;
        }

        public String getSecret() {
            return secret;
        }

        public void setSecret(String secret) {
            this.secret = secret;
        }

        public int getSessionTimeoutInSeconds() {
            return sessionTimeoutInSeconds;
        }

        public void setSessionTimeoutInSeconds(int sessionTimeoutInSeconds) {
            this.sessionTimeoutInSeconds = sessionTimeoutInSeconds;
        }

        public String getCookieDomain() {
            return cookieDomain;
        }

        public void setCookieDomain(String cookieDomain) {
            this.cookieDomain = cookieDomain;
        }
    }

    public static class SignatureVerification {
        /**
```

```
        * Maximum refresh rate for public keys in ms.
        * We won't fetch new public keys any faster than that to avoid spamming UAA in case
        * we receive a lot of "illegal" tokens.
        */
       private long publicKeyRefreshRateLimit = 10 * 1000L;
       /**
        * Maximum TTL for the public key in ms.
        * The public key will be fetched again from UAA if it gets older than that.
        * That way, we make sure that we get the newest keys always in case they are updated there.
        */
       private long ttl = 24 * 60 * 60 * 1000L;
       /**
        * Endpoint where to retrieve the public key used to verify token signatures.
        */
       private String publicKeyEndpointUri = "http://uaa/oauth/token_key";

       public long getPublicKeyRefreshRateLimit() {
          return publicKeyRefreshRateLimit;
       }

       public void setPublicKeyRefreshRateLimit(long publicKeyRefreshRateLimit) {
          this.publicKeyRefreshRateLimit = publicKeyRefreshRateLimit;
       }

       public long getTtl() {
          return ttl;
       }

       public void setTtl(long ttl) {
          this.ttl = ttl;
       }

       public String getPublicKeyEndpointUri() {
          return publicKeyEndpointUri;
       }

       public void setPublicKeyEndpointUri(String publicKeyEndpointUri) {
          this.publicKeyEndpointUri = publicKeyEndpointUri;
       }
    }
}
OAuth2JwtAccessTokenConverter
package org.events.salvatory.config.oauth2;

import org.events.salvatory.security.oauth2.OAuth2SignatureVerifierClient;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.jwt.crypto.sign.SignatureVerifier;
import org.springframework.security.oauth2.common.exceptions.InvalidTokenException;
import org.springframework.security.oauth2.provider.token.store.JwtAccessTokenConverter;
import org.springframework.security.oauth2.provider.OAuth2Authentication;

import java.util.Map;

/**
 * Improved {@link JwtAccessTokenConverter} that can handle lazy fetching of public verifier keys.
 */
public class OAuth2JwtAccessTokenConverter extends JwtAccessTokenConverter {
    private final Logger log = LoggerFactory.getLogger(OAuth2JwtAccessTokenConverter.class);

    private final OAuth2Properties oAuth2Properties;
    private final OAuth2SignatureVerifierClient signatureVerifierClient;
    /**
     * When did we last fetch the public key?
     */
    private long lastKeyFetchTimestamp;
```

```
    public OAuth2JwtAccessTokenConverter(OAuth2Properties oAuth2Properties, OAuth2SignatureVerifierClient
signatureVerifierClient) {
        this.oAuth2Properties = oAuth2Properties;
        this.signatureVerifierClient = signatureVerifierClient;
        tryCreateSignatureVerifier();
    }

    /**
     * Try to decode the token with the current public key.
     * If it fails, contact the OAuth2 server to get a new public key, then try again.
     * We might not have fetched it in the first place or it might have changed.
     *
     * @param token the JWT token to decode.
     * @return the resulting claims.
     * @throws InvalidTokenException if we cannot decode the token.
     */
    @Override
    protected Map<String, Object> decode(String token) {
        try {
            //check if our public key and thus SignatureVerifier have expired
            long ttl = oAuth2Properties.getSignatureVerification().getTtl();
            if (ttl > 0 && System.currentTimeMillis() - lastKeyFetchTimestamp > ttl) {
                throw new InvalidTokenException("public key expired");
            }
            return super.decode(token);
        } catch (InvalidTokenException ex) {
            if (tryCreateSignatureVerifier()) {
                return super.decode(token);
            }
            throw ex;
        }
    }

    /**
     * Fetch a new public key from the AuthorizationServer.
     *
     * @return true, if we could fetch it; false, if we could not.
     */
    private boolean tryCreateSignatureVerifier() {
        long t = System.currentTimeMillis();
        if (t - lastKeyFetchTimestamp < oAuth2Properties.getSignatureVerification().getPublicKeyRefreshRateLimit()) {
            return false;
        }
        try {
            SignatureVerifier verifier = signatureVerifierClient.getSignatureVerifier();
            if (verifier != null) {
                setVerifier(verifier);
                lastKeyFetchTimestamp = t;
                log.debug("Public key retrieved from OAuth2 server to create SignatureVerifier");
                return true;
            }
        } catch (Throwable ex) {
            log.error("could not get public key from OAuth2 server to create SignatureVerifier", ex);
        }
        return false;
    }
    /**
     * Extract JWT claims and set it to OAuth2Authentication decoded details.
     * Here is how to get details:
     *
     * <pre>
     * <code>
     *  SecurityContext securityContext = SecurityContextHolder.getContext();
     *  Authentication authentication = securityContext.getAuthentication();
     *  if (authentication != null) {
     *      Object details = authentication.getDetails();
     *      if (details instanceof OAuth2AuthenticationDetails) {
```

```
     *         Object decodedDetails = ((OAuth2AuthenticationDetails) details).getDecodedDetails();
     *         if (decodedDetails != null &amp;amp;&amp;amp; decodedDetails instanceof Map) {
     *           String detailFoo = ((Map) decodedDetails).get("foo");
     *         }
     *     }
     *  }
     * </code>
     *  </pre>
     * @param claims OAuth2JWTToken claims.
     * @return {@link OAuth2Authentication}.
     */
    @Override
    public OAuth2Authentication extractAuthentication(Map<String, ?> claims) {
        OAuth2Authentication authentication = super.extractAuthentication(claims);
        authentication.setDetails(claims);
        return authentication;
    }
}
```